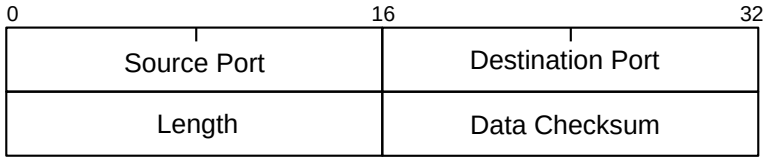
16 UDP TRANSPORT

The standard transport protocols riding above the IP layer are **TCP** and **UDP**. As we saw in Chapter 1, UDP provides simple datagram delivery to remote sockets, that is, to ⟨host,port⟩ pairs. TCP provides a much richer functionality for sending data, but requires that the remote socket first be *connected*. In this chapter, we start with the much-simpler UDP, including the UDP-based Trivial File Transfer Protocol.

We also review some fundamental issues any transport protocol must address, such as lost final packets and packets arriving late enough to be subject to misinterpretation upon arrival. These fundamental issues will be equally applicable to TCP connections.

16.1 User Datagram Protocol – UDP

RFC 1122 refers to UDP as "almost a null protocol"; while that is something of a harsh assessment, UDP is indeed fairly basic. The two features it adds beyond the IP layer are **port numbers** and a **checksum**. The UDP header consists of the following:



The port numbers are what makes UDP into a real transport protocol: with them, an application can now connect to an individual server *process* (that is, the process "owning" the port number in question), rather than simply to a host.

UDP is **unreliable**, in that there is no UDP-layer attempt at timeouts, acknowledgment and retransmission; applications written for UDP must implement these. As with TCP, a UDP ⟨host,port⟩ pair is known as a **socket** (though UDP ports are considered a separate namespace from TCP ports). UDP is also **unconnected**, or stateless; if an application has opened a port on a host, any other host on the Internet may deliver packets to that ⟨host,port⟩ socket without preliminary negotiation.

> An old bit of Internet humor about UDP's unreliability has it that *if I send you a UDP joke, you might not get it*.

UDP packets use the 16-bit Internet **checksum** (*7.4 Error Detection*) on the data. While it is seldom done today, the checksum can be disabled by setting the checksum field to the all-0-bits value, which never occurs as an actual ones-complement sum. The UDP checksum covers the UDP header, the UDP data and also a "pseudo-IP header" that includes the source and destination IP addresses (and also a duplicate copy of the UDP-header `length` field). If a NAT router rewrites an IP address or port, the UDP checksum must be updated.

UDP packets can be dropped due to queue overflows either at an intervening router or at the receiving host. When the latter happens, it means that packets are arriving faster than the receiver can process them. Higher-level protocols that define ACK packets (*eg* UDP-based RPC, below) typically include some form of **flow control** to prevent this.

UDP is popular for "local" transport, confined to one LAN. In this setting it is common to use UDP as the transport basis for a **Remote Procedure Call**, or RPC, protocol. The conceptual idea behind RPC is that one host invokes a procedure on another host; the parameters and the return value are transported back and forth by UDP. We will consider RPC in greater detail below, in *16.5 Remote Procedure Call (RPC)*; for now, the point of UDP is that on a local LAN we can fall back on rather simple mechanisms for timeout and retransmission.

UDP is well-suited for "request-reply" semantics beyond RPC; one can use TCP to send a message and get a reply, but there is the additional overhead of setting up and tearing down a connection. DNS uses UDP, largely for this reason. However, if there is any chance that a sequence of request-reply operations will be performed in short order then TCP may be worth the overhead.

UDP is also popular for **real-time** transport; the issue here is head-of-line blocking. If a TCP packet is lost, then the receiving *host* queues any later data until the lost data is retransmitted successfully, which can take several RTTs; there is no option for the receiving *application* to request different behavior. UDP, on the other hand, gives the receiving application the freedom simply to ignore lost packets. This approach is very successful for voice and video, which are **loss-tolerant** in that small losses simply degrade the received signal slightly, but **delay-intolerant** in that packets arriving too late for playback might as well not have arrived at all. Similarly, in a computer game a lost position update is moot after any subsequent update. Loss tolerance is the reason the **Real-time Transport Protocol**, or RTP, is built on top of UDP rather than TCP. It is common for VoIP telephone calls to use RTP and UDP. See also the NoTCP Manifesto.

There is a dark side to UDP: it is sometimes the protocol of choice in flooding attacks on the Internet, as it is easy to send UDP packets with spoofed source address. See the Internet Draft draft-byrne-opsec-udp-advisory. That said, it is not especially hard to send TCP connection-request (SYN) packets with spoofed source address. It is, however, quite difficult to get TCP source-address spoofing to work for long enough that data is delivered to an application process; see *18.3.1 ISNs and spoofing*.

UDP also sometimes enables what are called **traffic amplification** attacks: the attacker sends a small message to a server, with spoofed source address, and the server then responds to the spoofed address with a much larger response message. This creates a larger volume of traffic to the victim than the attacker would be able to generate directly. One approach is for the server to limit the size of its response – ideally to the size of the client's request – until it has been able to verify that the client actually receives packets sent to its claimed IP address. QUIC uses this approach; see *18.15.4.4 Connection handshake and TLS encryption*.

16.1.1 QUIC

Sometimes UDP is used simply because it allows new or experimental protocols to run entirely as user-space applications; no kernel updates are required, as would be the case with TCP changes. Google has created a protocol named **QUIC** (Quick UDP Internet Connections, chromium.org/quic) in this category, rather specifically to support the HTTP protocol. QUIC can in fact be viewed as a transport protocol specifically tailored to HTTPS: HTTP plus TLS encryption (*29.5.2 TLS*).

QUIC also takes advantage of UDP's freedom from head-of-line blocking. For example, one of QUIC's goals includes supporting multiplexed streams in a single connection (*eg* for the multiple components of a

web page). A lost packet blocks its own stream until it is retransmitted, but the other streams can continue without waiting. An early version of QUIC supported error-correcting codes (*7.4.2 Error-Correcting Codes*); this is another feature that would be difficult to add to TCP.

In many cases QUIC eliminates the initial RTT needed for setting up a TCP connection, allowing data delivery with the very first packet. This usually this requires a recent previous connection, however, as otherwise accepting data in the first packet opens the recipient up to certain spoofing attacks. Also, QUIC usually eliminates the second (and maybe third) RTT needed for negotiating TLS encryption (*29.5.2 TLS*).

QUIC provides support for advanced congestion control, currently (2014) including a UDP analog of TCP CUBIC (*22.15 TCP CUBIC*). QUIC does this at the application layer but new congestion-control mechanisms within TCP often require client operating-system changes even when the mechanism lives primarily at the server end. (QUIC may require kernel support to make use of ECN congestion feedback, *21.5.3 Explicit Congestion Notification (ECN)*, as this requires setting bits in the IP header.) QUIC represents a promising approach to using UDP's flexibility to support innovative or experimental transport-layer features.

One downside of QUIC is its nonstandard programming interface, but note that Google can (and does) achieve widespread web utilization of QUIC simply by distributing the client side in its Chrome browser. Another downside, more insidious, is that QUIC breaks the "social contract" that everyone should use TCP so that everyone is on the same footing regarding congestion. It turns out, though, that TCP users are *not* in fact all on the same footing, as there are now multiple TCP variants (*22 Newer TCP Implementations*). Furthermore, QUIC is *supposed* to compete fairly with TCP. Still, QUIC does open an interesting can of worms.

Because many of the specific features of QUIC were chosen in response to perceived difficulties with TCP, we will explore the protocol's details after introducing TCP, in *18.15.4 QUIC Revisited*.

16.1.2 DCCP

The Datagram Congestion Control Protocol, or **DCCP**, is another transport protocol build atop UDP, preserving UDP's fundamental tolerance to packet loss. It is outlined in **RFC 4340**. DCCP adds a number of TCP-like features to UDP; for our purposes the most significant are connection setup and teardown (see *18.15.3 DCCP*) and TCP-like congestion management (see *21.3.3 DCCP Congestion Control*).

DCCP data packets, while numbered, are delivered to the application in order of arrival rather than in order of sequence number. DCCP also adds acknowledgments to UDP, but in a specialized form primarily for congestion control. There is no assumption that unacknowledged data packets will ever be retransmitted; that decision is entirely up to the application. Acknowledgments can acknowledge single packets or, through the DCCP acknowledgment-vector format, all packets received in a range of recent sequence numbers (SACK TCP, *19.6 Selective Acknowledgments (SACK)*, also supports this).

DCCP does support reliable delivery of control packets, used for connection setup, teardown and option negotiation. Option negotiation can occur at any point during a connection.

DCCP packets include not only the usual application-specific UDP port numbers, but also a 32-bit **service code**. This allows finer-grained packet handling as it unambiguously identifies the processing requested by an incoming packet. The use of service codes also resolves problems created when applications are forced to use nonstandard port numbers due to conflicts.

DCCP is specifically intended to run in in the operating-system kernel, rather than in user space. This is because the ECN congestion-feedback mechanism (*21.5.3 Explicit Congestion Notification (ECN)*) requires

setting flag bits in the IP header, and most kernels do not allow user-space applications to do this.

16.1.3 UDP Simplex-Talk

One of the early standard examples for socket programming is simplex-talk. The client side reads lines of text from the user's terminal and sends them over the network to the server; the server then displays them on its terminal. The server does not acknowledge anything sent to it, or in fact send any response to the client at all. "Simplex" here refers to the one-way nature of the flow; "duplex talk" is the basis for Instant Messaging, or IM.

Even at this simple level we have some details to attend to regarding the data protocol: we assume here that the lines are sent *with* a trailing end-of-line marker. In a world where different OS's use different end-of-line marks, including them in the transmitted data can be problematic. However, when we get to the TCP version, if arriving packets are queued for any reason then the embedded end-of-line character will be the only thing to separate the arriving data into lines.

As with almost every Internet protocol, the server side must select a port number, which with the server's IP address will form the **socket address** to which clients connect. Clients must discover that port number or have it written into their application code. Clients too will *have* a port number, but it is largely invisible.

On the server side, simplex-talk must do the following:

- ask for a designated port number
- create a **socket**, the sending/receiving endpoint
- **bind** the socket to the socket address, if this is not done at the point of socket creation
- receive packets sent to the socket
- for each packet received, print its sender and its content

The client side has a similar list:

- **look up** the server's IP address, using DNS
- create an "anonymous" socket; we don't care what the client's port number is
- read a line from the terminal, and send it to the socket address ⟨server_IP,port⟩

16.1.3.1 The Server

We will start with the server side, presented here in Java. The Java socket implementation is based mostly on the BSD socket library, *1.16 Berkeley Unix*. We will use port 5432; this can easily be changed if, for example, on startup an error message like "cannot create socket with port 5432" appears. The port we use here, 5432, has also been adopted by PostgreSQL for TCP connections. (The client, of course, would also need to be changed.)

Java `DatagramPacket` type

Java `DatagramPacket` objects contain the packet data and the ⟨IP_address,port⟩ source or destination. Packets themselves combine both data and address, of course, but nonetheless combining these in a single programming-language object is not an especially common design choice. The original BSD socket library implemented data and address as separate parameters, and many other languages have followed that precedent. A case can be made that the Java approach violates the single-responsibility principle, because data and address are so often handled separately.

The socket-creation and port-binding operations are combined into the single operation `new DatagramSocket(destport)`. Once created, this socket will receive packets from any host that addresses a packet to it; there is no need for preliminary connection. In the original BSD socket library, a socket is created with `socket()` and bound to an address with the separate operation `bind()`.

The server application needs no parameters; it just starts. (That said, we could make the port number a parameter, to allow easy change.) The server accepts both IPv4 and IPv6 connections; we return to this below.

Though it plays no role in the protocol, we will also have the server time out every 15 seconds and display a message, just to show how this is done. Implementations of real UDP protocols essentially *always* must arrange when attempting to receive a packet to time out after a certain interval with no response.

The file below is at udp_stalks.java.

```
/* simplex-talk server, UDP version */

import java.net.*;
import java.io.*;

public class stalks {

    static public int destport = 5432;
    static public int bufsize = 512;
    static public final int timeout = 15000; // time in milliseconds

    static public void main(String args[]) {
        DatagramSocket s;               // UDP uses DatagramSockets

        try {
            s = new DatagramSocket(destport);
        }
        catch (SocketException se) {
            System.err.println("cannot create socket with port " + destport);
            return;
        }
        try {
            s.setSoTimeout(timeout);       // set timeout in milliseconds
        } catch (SocketException se) {
            System.err.println("socket exception: timeout not set!");
        }

        // create DatagramPacket object for receiving data:
        DatagramPacket msg = new DatagramPacket(new byte[bufsize], bufsize);
```

(continues on next page)

(continued from previous page)

```
        while(true) { // read loop
            try {
                msg.setLength(bufsize);  // max received packet size
                s.receive(msg);          // the actual receive operation
                System.err.println("message from <" +
                   msg.getAddress().getHostAddress() + "," + msg.getPort() +
↪">");
            } catch (SocketTimeoutException ste) {    // receive() timed out
                System.err.println("Response timed out!");
                continue;
            } catch (IOException ioe) {                // should never happen!
                System.err.println("Bad receive");
                break;
            }

            String str = new String(msg.getData(), 0, msg.getLength());
            System.out.print(str);        // newline must be part of str
        }
        s.close();
    } // end of main
}
```

16.1.3.2 UDP and IP addresses

The server line `s = new DatagramSocket(destport)` creates a `DatagramSocket` object **bound** to the given port. If a host has multiple IP addresses (that is, is multihomed), packets sent to that port to any of those IP addresses will be delivered to the socket, including `localhost` (and in fact all IPv4 addresses between 127.0.0.1 and 127.255.255.255) and the subnet broadcast address (*eg* 192.168.1.255). If a client attempts to connect to the subnet broadcast address, multiple servers may receive the packet (in this we are perhaps fortunate that the stalk server does not reply).

Alternatively, we could have used

```
s = new DatagramSocket(int port, InetAddress local_addr)
```

in which case only packets sent to the host and port through the host's specific IP address `local_addr` would be delivered. It does not matter here whether IP forwarding on the host has been enabled. In the original C socket library, this binding of a port to (usually) a server socket was done with the `bind()` call. To allow connections via any of the host's IP addresses, the special IP address `INADDR_ANY` is passed to `bind()`.

When a host has multiple IP addresses, the BSD socket library and its descendents do not appear to provide a way to find out to which these an arriving UDP packet was actually sent (although it is supposed to, according to **RFC 1122**, §4.1.3.5). Normally, however, this is not a major difficulty. If a host has only one interface on an actual network (*ie* not counting loopback), and only one IP address for that interface, then any remote clients must send to that interface and address. Replies (if any, which there are not with stalk) will also come from that address.

Multiple interfaces do not necessarily create an ambiguity either; the easiest such case to experiment with

involves use of the loopback and Ethernet interfaces (though one would need to use an application that, unlike stalk, sends replies). If these interfaces have respective IPv4 addresses 127.0.0.1 and 192.168.1.1, and the client is run on the same machine, then connections to the server application sent to 127.0.0.1 will be answered from 127.0.0.1, and connections sent to 192.168.1.1 will be answered from 192.168.1.1. The IP layer sees these as different subnets, and fills in the IP source-address field according to the appropriate subnet. The same applies if multiple Ethernet interfaces are involved, or if a single Ethernet interface is assigned IP addresses for two different subnets, *eg* 192.168.**1**.1 and 192.168.**2**.1.

Life is slightly more complicated if a single interface is assigned multiple IP addresses on the *same* subnet, *eg* 192.168.1.**1** and 192.168.1.**2**. Regardless of which address a client sends its request to, the server's reply will generally always come from one designated address for that subnet, *eg* 192.168.1.1. Thus, it is possible that a *legitimate UDP reply will come from a different IP address than that to which the initial request was sent*.

If this behavior is not desired, one approach is to create multiple server sockets, and to bind each of the host's network IP addresses to a different server socket.

The fact that the IP layer usually chooses the source address adds a slight wrinkle to the discussion of network protocol layers at *1.15 IETF and OSI*. The classic "encapsulation" model would suggest that the UDP layer writes the UDP header and then passes the packet (and destination IP address) to the IP layer, which then writes the IP header and passes the packet in turn down to the LAN layer. But this cannot work quite as described, because, if the IP source address is seen as supplied by the IP layer, then would not be available at the time the UDP-header checksum field is first filled in. Checksums are messy, and real implementations simply blur the layering "rules": typically the UDP layer asks the IP layer for early determination of the IP source address. The situation is further complicated by the fact that nowadays the bulk of the checksum calculation is often performed at the LAN layer, by the LAN hardware; see *17.5 TCP Offloading*.

16.1.3.3 The Client

Next is the Java client version udp_stalkc.java. The client – any client – *must* provide the name of the host to which it wishes to send; as with the port number this can be hard-coded into the application but is more commonly specified by the user. The version here uses host `localhost` as a default but accepts any other hostname as a command-line argument. The call to `InetAddress.getByName(desthost)` invokes the DNS system, which looks up name desthost and, if successful, returns an IP address. (`InetAddress.getByName()` also accepts addresses in numeric form, *eg* "127.0.0.1", in which case DNS is not necessary.) When we create the socket we do not designate a port in the call to `new DatagramSocket()`; this means any port will do for the client. When we create the `DatagramPacket` object, the first parameter is a zero-length array as the actual data array will be provided within the loop.

A certain degree of messiness is introduced by the need to create a `BufferedReader` object to handle terminal input.

```
// simplex-talk CLIENT in java, UDP version

import java.net.*;
import java.io.*;

public class stalkc {
```

(continues on next page)

(continued from previous page)

```
    static public BufferedReader bin;
    static public int destport = 5432;
    static public int bufsize = 512;

    static public void main(String args[]) {
        String desthost = "localhost";
        if (args.length >= 1) desthost = args[0];

        bin = new BufferedReader(new InputStreamReader(System.in));

        InetAddress dest;
        System.err.print("Looking up address of " + desthost + "...");
        try {
            dest = InetAddress.getByName(desthost);        // DNS query
        }
        catch (UnknownHostException uhe) {
            System.err.println("unknown host: " + desthost);
            return;
        }
        System.err.println(" got it!");

        DatagramSocket s;
        try {
            s = new DatagramSocket();
        }
        catch(IOException ioe) {
            System.err.println("socket could not be created");
            return;
        }

        System.err.println("Our own port is " + s.getLocalPort());

        DatagramPacket msg = new DatagramPacket(new byte[0], 0, dest,␣
↪destport);

        while (true) {
            String buf;
            int slen;
            try {
                buf = bin.readLine();
            }
            catch (IOException ioe) {
                System.err.println("readLine() failed");
                return;
            }

            if (buf == null) break;       // user typed EOF character

            buf = buf + "\n";             // append newline character
            slen = buf.length();
            byte[] bbuf = buf.getBytes();
```

(continues on next page)

(continued from previous page)

```
            msg.setData(bbuf);
            msg.setLength(slen);

            try {
                s.send(msg);
            }
            catch (IOException ioe) {
                System.err.println("send() failed");
                return;
            }
        } // while
        s.close();
    }
}
```

The default value of `desthost` here is `localhost`; this is convenient when running the client and the server on the same machine, in separate terminal windows.

All packets are sent to the ⟨dest,destport⟩ address specified in the initialization of `msg`. Alternatively, we could have called `s.connect(dest,destport)`. This causes nothing to be sent over the network, as UDP is connectionless, but locally marks the socket `s` allowing it to send only to ⟨dest,destport⟩. In Java we still have to embed the destination address in every `DatagramPacket` we `send()`, so this offers no benefit, but in other languages this can simplify subsequent sending operations.

Like the server, the client works with both IPv4 and IPv6. The `InetAddress` object `dest` in the server code above can hold either IPv4 or IPv6 addresses; `InetAddress` is the base class with child classes `Inet4Address` and `Inet6Address`. If the client and server can communicate at all via IPv6 and if the value of `desthost` supplied to the client is an IPv6-only name, then `dest` will be an `Inet6Address` object and IPv6 will be used.

For example, if the client is invoked from the command line with `java stalkc ip6-localhost`, and the name `ip6-localhost` resolves to the IPv6 loopback address `::1`, the client will send its packets to an stalk server on the same host using IPv6 (and the loopback interface).

If greater IPv4-versus-IPv6 control is desired, one can replace the `getByName()` call with the following, where `dests` now has type `InetAddress[]`:

```
dests = InetAddress.getAllByName(desthost);
```

This returns an array of all addresses associated with the given name. One can then find the IPv6 addresses by searching this array for addresses `addr` for which `addr instanceof Inet6Address`.

For non-Java languages, IP-address objects often have an `AddressFamily` attribute that can be used to determine whether an address is IPv4 or IPv6. See also *12.4 Using IPv6 and IPv4 Together*.

Finally, here is a simple python version of the client, udp_stalkc.py.

```
#!/usr/bin/python3

from socket import *
```

(continues on next page)

(continued from previous page)

```
from sys import argv

portnum = 5432

def talk():
        rhost = "localhost"
        if len(argv) > 1:
            rhost = argv[1]
        print("Looking up address of " + rhost + "...", end="")
        try:
            dest = gethostbyname(rhost)
        except (GAIerror, herror) as mesg:      # GAIerror: error in␣
↪gethostbyname()
            errno,errstr=mesg.args
            print("\n   ", errstr);
            return;
        print("got it: " + dest)
        addr=(dest, portnum)                    # a socket address
        s = socket(AF_INET, SOCK_DGRAM)
        s.settimeout(1.5)               # we don't actually need to set␣
↪timeout here
        while True:
            try:
                buf = input("> ")
            except:
                break
            s.sendto(bytes(buf + "\n", 'ascii'), addr)

talk()
```

Why not C?

While C is arguably the most popular language for network programming, it does not support IP addresses and other network objects as first-class types, and so we omit it here. But see *28.2.2 An Actual Stack-Overflow Example* and *29.5.3 A TLS Programming Example* for TCP-based C versions of stalk-like programs. (The problem is not entirely C's fault; a network address might be an IPv4 address or an IPv6 address (or even a "named pipe" address); these objects are of different sizes and so addresses must be handled by reference, which is awkward in C.)

To experiment with these on a single host, start the server in one window and one or more clients in other windows. One can then try the following:

- have two clients simultaneously running, and sending alternating messages to the same server
- invoke the client with the external IP address of the server in dotted-decimal, *eg* 10.0.0.3 (note that `localhost` is 127.0.0.1)
- run the java and python clients simultaneously, sending to the same server
- run the server on a different host (*eg* a virtual host or a neighboring machine)

- invoke the client with a nonexistent hostname

One can also use `netcat`, below, as a client, though `netcat` as a server will not work for the multiple-client experiments.

Note that, depending on the DNS server, the last one may not actually fail. When asked for the DNS name of a nonexistent host such as zxqzx.org, many ISPs will return the IP address of a host running a web server hosting an error/search/advertising page (usually their own). This makes some modicum of sense when attention is restricted to web searches, but is annoying if it is not, as it means non-web applications have no easy way to identify nonexistent hosts.

Simplex-talk will work if the server is on the public side of a NAT firewall. No server-side packets need to be delivered to the client! But if the other direction works, something is very wrong with the firewall.

16.1.4 `netcat`

The versatile `netcat` utility (also sometimes spelled `nc`) utility enables sending and receiving of individual UDP (and TCP) packets; we can use it to substitute for the stalk client, or, with a limitation, the server. (The `netcat` utility, unlike stalk, supports bidirectional communication.)

The `netcat` utility is available for Windows, Linux and Macintosh systems, in both binary and source forms, from a variety of places and in something of a variety of versions. The classic version is available from sourceforge.net/projects/nc110; a newer implementation is ncat. The Wikipedia page has additional information.

As with stalk, `netcat` sends the final end-of-line marker along with its data. The `-u` flag is used to request UDP. To send to port 5432 on `localhost` using UDP, like an stalk client, the command is

```
netcat -u localhost 5432
```

One can then type multiple lines that should all be received by a running stalk server. If desired, the source port can be specified with the `-p` option; *eg* `netcat -u -p 40001 localhost 5432`.

To act as an stalk *server*, we need the `-l` option to ask `netcat` to listen instead of sending:

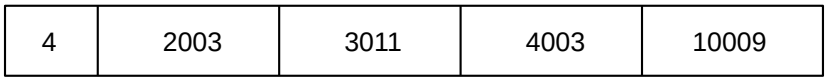
```
netcat -l -u 5432
```

One can then send lines using `stalkc` or `netcat` in client mode. However, once `netcat` in server mode receives its first UDP packet, it will not accept later UDP packets from different sources (some versions of `netcat` have a `-k` option to allow this for TCP, but not for UDP). (This situation arises because `netcat` makes use of the `connect()` call on the server side as well as the client, after which the server can only send to and receive from the socket address to which it has connected. This simplifies bidirectional communication. Often, UDP `connect()` is called only by the client, if at all. See the paragraph about `connect()` following the Java `stalkc` code in *16.1.3.3 The Client*.)

16.1.5 Binary Data

In the stalk example above, the client sent strings to the server. However, what if we are implementing a protocol that requires us to send *binary* data? Or designing such a protocol? The client and server will now have to agree on how the data is to be **encoded**.

As an example, suppose the client is to send to the server a list of 32-bit integers, organized as follows. The length of the list is to occupy the first two bytes; the remainder of the packet contains the consecutive integers themselves, four bytes each, as in the diagram:



packet data layout

The client needs to create the byte array organized as above, and the server needs to extract the values. (The inclusion of the list length as a `short int` is not really necessary, as the receiver will be able to infer the list length from the packet size, but we want to be able to illustrate the encoding of both `int` and `short int` values.)

The protocol also needs to define how the integers themselves are laid out. There are two common ways to represent a 32-bit integer as a sequence of four bytes. Consider the integer 0x01020304 = $1 \times 256^3 + 2 \times 256^2 + 3 \times 256 + 4$. This can be encoded as the byte sequence [1,2,3,4], known as **big-endian** encoding, or as [4,3,2,1], known as **little-endian** encoding; the former was used by early IBM mainframes and the latter is used by most Intel processors. (We are assuming here that both architectures represent signed integers using twos-complement; this is now universal but was not always.)

To send 32-bit integers over the network, it is certainly possible to tag the data as big-endian or little-endian, or for the endpoints to negotiate the encoding. However, by far the most common approach on the Internet – at least below the application layer – is to follow the convention of **RFC 1700** and use big-endian encoding exclusively; big-endian encoding has since come to be known as "network byte order".

How one converts from "host byte order" to "network byte order" is language-dependent. It must always be done, even on big-endian architectures, as code may be recompiled on a different architecture later.

In Java the byte-order conversion is generally combined with the process of conversion from `int` to `byte[]`. The client will use a `DataOutputStream` class to support the writing of the binary values to an output stream, through methods such as `writeInt()` and `writeShort()`, together with a `ByteArrayOutputStream` class to support the conversion of the output stream to type `byte[]`. The code below assumes the list of integers is initially in an `ArrayList<Integer>` named `theNums`.

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);
try {
    dos.writeShort(theNums.size());
    for (int n : theNums) {
        dos.writeInt(n);
    }
} catch (IOException ioe) { /* exception handling */ }

byte[] bbuf = baos.toByteArray();
msg.setData(bbuf);                          // msg is the DatagramPacket␣
↪object to be sent
```

The server then needs to to the reverse; again, `msg` is the arriving DatagramPacket. The code below simply calculates the sum of the 32-bit integers in `msg`:

```
ByteArrayInputStream bais = new ByteArrayInputStream(msg.getData(), 0, msg.
↪getLength());
DataInputStream dis = new DataInputStream(bais);
int sum = 0;
try {
    int count = dis.readShort();
    for (int i=0; i<count; i++) {
        sum += dis.readInt();
    }
} catch (IOException ioe) {  /* more exception handling */  }
```

A version of simplex-talk for lists of integers can be found in client saddc.java and server sadds.java. The client reads from the command line a list of character-encoded integers (separated by whitespace), constructs the binary encoding as above, and sends them to the server; the server prints their sum. Port 5434 is used; this can be changed if necessary.

In the C language, we can simply allocate a `char[]` of the appropriate size and write the network-byte-order values directly into it. Conversion to network byte order and back is done with the following library calls:

- `htonl()`: host-to-network conversion for long (32-bit) integers
- `ntohl()`: network-to-host conversion for long integers
- `htons()`: host-to-network conversion for short (16-bit) integers
- `ntohs()`: network-to-host conversion for short integers

A certain amount of casting between `int *` and `char *` is also necessary. As both casting and byte-order conversions are error-prone, it is best if all conversions are made in a block, just after a packet arrives or just before it is sent, rather than on demand throughout the program.

In general, the designer of a protocol needs to select an unambiguous format for all binary data; protocol-defining RFCs always include such format details. This can be a particular issue for floating-point data, for which two formats can have the same endianness but still differ, *eg* in normalization or the size of the exponent field. Formats for structured data, such as arrays, must also be spelled out; in the example above the list size was indicated by a length field but other options are possible.

The example above illustrates fixed-field-width encoding. Another possible option, using variable-length encoding, is ASN.1 using the Basic Encoding Rules (*26.6 ASN.1 Syntax and SNMP*); fixed-field encoding sometimes becomes cumbersome as data becomes more hierarchical.

At the application layer, the use of non-binary encodings is common, though binary encodings continue to remain common as well. Two popular formats using human-readable unicode strings for data encoding are ASN.1 with its XML Encoding Rules and JSON. While the latter format originated with JavaScript, it is now widely supported by many other languages.

16.2 Trivial File Transport Protocol, TFTP

We now introduce a real protocol based on UDP: the Trivial File Transport Protocol, or TFTP. While TFTP supports file transfers in both directions, we will restrict attention to the more common case where the client

requests a file from the server. TFTP does not support a mechanism for authentication; any requestable files are available to anyone. In this TFTP does not differ from basic web browsing; as with web servers, a TFTP file server must ensure that requests are disallowed if the file – for example `../../../etc/passwd` – is not within a permitted directory.

Because TFTP is UDP-based, and clients can be implemented very compactly, it is well-suited to the downloading of startup files to very compact systems, including diskless systems. Because it uses stop-and-wait, often uses a fixed timeout interval, and offers limited security, TFTP is typically confined to internal use within a LAN.

Although TFTP is a very simple protocol, for correct operation it must address several fundamental transport issues; these are discussed in detail in the following section. TFTP is presented here partly as a way to introduce these transport issues; we will later return to these same issues in the context of TCP (*18.4 Anomalous TCP scenarios*).

TFTP, documented first in **RFC 783** and updated in **RFC 1350**, has five packet types:

- Read ReQuest, RRQ, containing the filename and a text/binary indication
- Write ReQuest, WRQ
- Data, containing a 16-bit block number and up to 512 bytes of data
- ACK, containing a 16-bit block number
- Error, for certain designated errors. All errors other than "Unknown Transfer ID" are cause for sender termination.

Data block numbering begins at 1; we will denote the packet with the Nth block of data as Data[N]. Acknowledgments contain the block number of the block being acknowledged; thus, ACK[N] acknowledges Data[N]. All blocks of data contain 512 bytes except the final block, which is identified *as* the final block by virtue of containing less than 512 bytes of data. If the file size was divisible by 512, the final block will contain 0 bytes of data. TFTP block numbers are 16 bits in length, and are not allowed to wrap around.

Because TFTP uses UDP (as opposed to TCP) it must take care of packetization itself, and thus must choose a block size small enough to avoid fragmentation (*9.4 Fragmentation*). While negotiation of the block size would have been possible, as is done by TCP's *18.6 Path MTU Discovery*, it would have added considerable complexity.

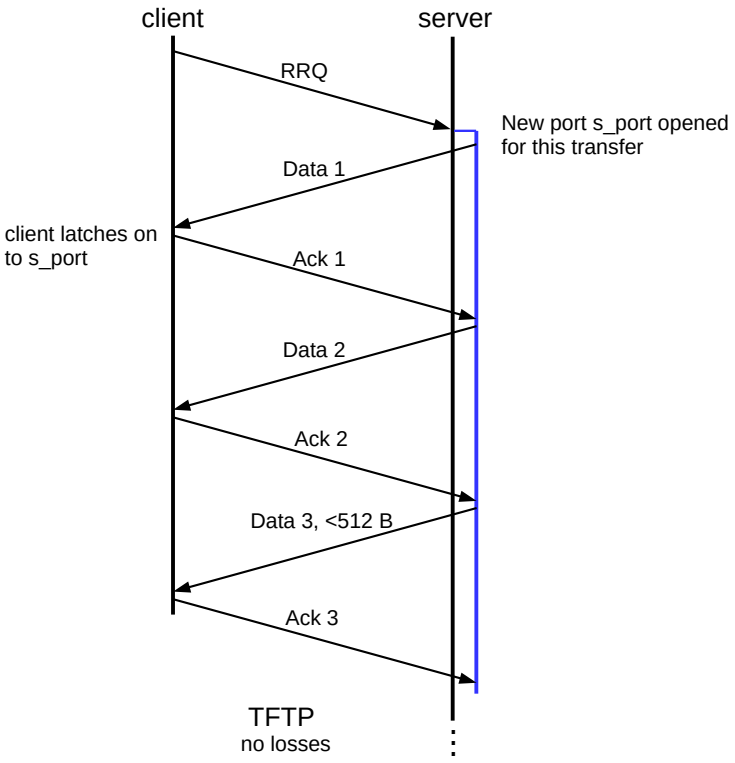
The TFTP server listens on UDP port 69 for arriving RRQ packets (and WRQ, though we will not consider those here). For each RRQ requesting a valid file, TFTP server implementations almost always create a separate process (or thread) to handle the transfer. That child process will then obtain an entirely new UDP port, which will be used for all further interaction with the client, at least for this particular transfer.

As we shall see below, this port change has significant functional implications in preventing old-duplicate packets, though for now we can justify it as making the implementer's life much easier. With the port change, the server child process responsible for the transfer has to interact with only one client; all arriving packets must have come from the client for which the child process was created (while it is possible for stray packets to arrive from other endpoints, the child process can ignore them). Without the port change, on the other hand, handling multiple concurrent transfers would be decidedly complicated: the server would have to sort out, for each arriving packet, which transfer it belonged to. Each transfer would have its own state information including block number, open file, and the time of the last successful packet. The port-change rule does have the drawback of preventing the use of TFTP through NAT firewalls.

In the absence of packet loss or other errors, TFTP file requests typically proceed as follows:

1. The client sends a RRQ to server port 69.
2. The server creates a child process, which obtains a new port, s_port, from the operating system.
3. The server child process sends Data[1] from s_port.
4. The client receives Data[1], and thus learns the value of s_port. The client will verify that each future Data[N] arrives from this same port.
5. The client sends ACK[1] (and all future ACKs) to the server's s_port.
6. The server child process sends Data[2], *etc*, each time waiting for the client ACK[N] before sending Data[N+1].
7. The transfer process stops when the server sends its final block, of size less than 512 bytes, and the client sends the corresponding ACK.

We will refer to the client's learning of s_port in step 3 as **latching on** to that port. Here is a diagram; the server child process (with new port s_port) is represented by the blue line at right.



We turn next to the complications introduced by taking packet losses and reordering into account.

16.3 Fundamental Transport Issues

The possibility of lost or delayed packets introduces several fundamental issues that any transport strategy must handle correctly for proper operation; we will revisit these in the context of TCP in *18.4 Anomalous*

TCP scenarios. The issues we will consider include

- old duplicate packets
- lost final ACK
- duplicated connection request
- reboots

In this section we will discuss these issues both in general and in particular how TFTP takes them into account.

16.3.1 Old Duplicate Packets

Perhaps the trickiest issue is **old duplicate packets**: packets from the past arriving quite late, but which are mistakenly accepted as current.

For a TFTP example, suppose the client chooses port 2000 and requests file "foo", and the server then chooses port 4000 for its child process. During this transfer, Data[2] gets duplicated (perhaps through timeout and retransmission) and one of the copies is greatly delayed. The other copy arrives on time, though, and the transfer concludes.

Now, more-or-less immediately after, the client initiates a second request, this time for file "bar". Fatefully, the client again chooses port 2000 and the server child process again chooses port 4000.

At the point in the second transfer when the client is waiting for Data[2] from file "bar", we will suppose the old-duplicate Data[2] from file "foo" finally shows up. There is nothing in the packet to indicate anything is amiss: the block number is correct, the destination port of 4000 ensures delivery to the current server child process, and the source port of 2000 makes the packet appear to be coming from the current client. The wrong Data[2] is therefore accepted as legitimate, and the file transfer is corrupted.

An old packet from a *previous* instance of the connection, as described above, is called an **external** old duplicate. An essential feature of the external case is that the connection is closed and then reopened a short time later, using the same port numbers at each end. As a connection is often *defined* by its endpoint port numbers (more precisely, its socket addresses), we refer to "reopening" the connection even if the second instance is completely unrelated. Two separate instances of a connection between the same socket addresses are sometimes known as **incarnations** of the connection, particularly in the context of TCP.

Old duplicates can also be **internal**, from an earlier point in the *same* connection instance. For example, if TFTP allowed its 16-bit block numbers to wrap around, then a very old Data[3] might be accepted in lieu of Data[3+2^{16}]. Internal old duplicates are usually prevented – or rendered improbable – by numbering the data, either by block or by byte, and using sufficiently many bits that wrap-around is unlikely. TFTP prevents *internal* old duplicates simply by not allowing its 16-bit block numbers to wrap around; this is effective, but limits the maximum file to 512B $\times$ (2^{16}–1), or about 32 MB. If we were not concerned with old duplicates, TFTP's stop-and-wait could make do with 1-bit sequence numbers (*8.5 Exercises*, exercise 8.5).

Random Ports?

RFC 1350 states "The TID's [port numbers] chosen for a connection should be randomly chosen, so that the probability that the same number is chosen twice in immediate succession is very low." A literal interpretation is that an implementation should choose a random 16-bit number and ask for that as its TID. But the author knows of no implementation that actually does this; all seem to create sockets (*eg* with Java's `DatagramSocket()`) and accept the port number assigned to the new socket by the operating system. That port number will not be "random" in the statistical sense, but *will* be very likely different from any recently used port. Should this be interpreted as noncompliance with the RFC? The author has no idea.

TFTP's defense against *external* old duplicates is based on requiring that both endpoints try to choose a different port for each separate transfer (**RFC 1350** states that each side should choose its port number "randomly"). As long as *either* endpoint succeeds in choosing a new port, external old duplicates cannot interfere; see exercise 7.0. If ports are chosen at random, the probability that both sides will chose the same pair of ports for the subsequent connection is around $1/2^{32}$; if ports are assigned by the operating system, there is an implicit assumption that the OS will not reissue the same port twice in rapid succession. If a noncompliant implementation on one side reuses its port numbers, TFTP transfers are protected as long as the other side chooses a new port, though the random probability of failure rises to $1/2^{16}$. Note that this issue represents a second, more fundamental reason for having the server choose a new port for each transfer, unrelated to making the implementer's life easier.

After enough time, port numbers will eventually be recycled, but we will assume old duplicates have a much smaller lifetime.

Both the external and internal old-duplicate scenarios assume that the old duplicate was sent earlier, but was somehow delayed in transit for an extended period of time, while later packets were delivered normally. Exactly how this might occur remains unclear; perhaps the least far-fetched scenario is the following:

- A first copy of the old duplicate was sent
- A routing error occurs; the packet is stuck in a routing loop
- An alternative path between the original hosts is restored, and the packet is retransmitted successfully
- Some time later, the packet stuck in the routing loop is released, and reaches its final destination

Another scenario involves a link in the path that supplies link-layer acknowledgment: the packet was sent once across the link, the link-layer ACK was lost, and so the packet was sent again. Some mechanism is still needed to delay one of the copies.

Most solutions to the old-duplicate problem assume *some* cap on just how late an old duplicate can be. In practical terms, TCP officially once took this time limit to be 60 seconds, but implementations now usually take it to be 30 seconds. Other protocols often implicitly adopt the TCP limit. Once upon a time, IP routers were expected to decrement a packet's TTL field by 1 for each second the router held the packet in its queue; in such a world, IP packets cannot be more than 255 seconds old.

It is also possible to prevent external old duplicates by including a **connection count** parameter in the transport or application header. For each consecutive connection, the connection count is incremented by (at least) 1. A separate connection-count value must be maintained by each side; if a connection-count value is ever lost, a suitable backup mechanism based on delay might be used. As an example, see *18.5 TCP Faster Opening*.

16.3.2 Lost Final ACK

In most protocols, most packets will be acknowledged. The final packet (almost always an ACK), however, cannot itself be acknowledged, as then it would not be the final packet. *Somebody* has to go last. This leaves some uncertainty on the part of the sender: did the last packet make it through, or not?

In the TFTP setting, suppose the server sends the final packet, Data[3]. The client receives it and sends ACK[3], and then exits as the transfer is done; however, the ACK[3] is lost.

The server will eventually time out and retransmit Data[3] again. However, the client is no longer there to receive the packet! The server will continue to timeout and retransmit the final Data packet until it gives up; it will never receive confirmation that the transfer succeeded.

More generally, if A sends a message to B and B replies with an acknowledgment that is delivered to A, then A and B both are both certain the message has been delivered successfully. B is *not* sure, however, that A knows this.

An alternative formulation of the lost-final-ACK problem is the **two-generals problem**. Two generals wish to agree on a time to attack, by exchanging messages. However, the generals *must* attack together, or not at all. Because some messages may be lost, neither side can ever be completely sure of the agreement. If the generals are Alice and Bob (*28.5.1 Alice and Bob*), the messages might look like this:

- Alice sends: Attack at noon
- Bob replies: Agreed (*ie* ACK)

After Bob receives Alice's message, both sides know that a noon attack has been proposed. After Bob's reply reaches Alice, both sides know that the message has been delivered. If Alice's message was an *order* to Bob, this would be sufficient.

But if Alice and Bob must cooperate, this is not quite enough: at the end of the exchange above, Bob does not know that Alice has received his reply; Bob might thus hesitate, fearing Alice might not know he's on board. Alice, aware of this possibility, might hesitate herself.

Alice might attempt to resolve this by acknowledging Bob's ACK:

- Alice replies: Ok, we're agreed on noon

But at this point Alice does not know if Bob has received this message. If Bob does not, Bob might still hesitate. Not knowing, Alice too might hesitate. There is no end. See [AEH75].

Mathematically, there is no perfect solution to the two-generals problem; the generals can never be certain they are in complete agreement to attack. Suppose, to the contrary, that a sequence of messages *did* bring certainty of agreement to both Alice and Bob. Let M_1, ..., M_n be the shortest possible such sequence; without loss of generality we may assume Alice sent M_n. Now consider what happens if this final message is lost. From Alice's perspective, there is no change at all, so Alice must still be certain Bob has agreed. However, the now-shorter sequence M_1, ..., M_{n-1} cannot also bring certainty to Bob, as this sequence has length less than n, the supposed minimum here. So Bob is *not* certain, and so Alice's certainty is misplaced.

In engineering terms, however, the probability of a misunderstanding can often be made vanishingly small. Typically, if Alice does not receive Bob's reply promptly, she will resend her message at regular timeout intervals, until she does receive an ACK. If Bob can count on this behavior, he can be reasonably sure that one of his ACKs must have made it back after enough time has elapsed.

For example, if Bob knows Alice will try a total of six times if she does not receive a response, and Bob only receives Alice's first two message instances, the fact that Alice appears to have stopped repeating her transmissions is reasonable evidence that she has received Bob's response. Alternatively, if each message/ACK pair has a 10% probability of failure, and Bob knows that Alice will retry her message up to six times over the course of a day, then by the end of the day Bob can conclude that the probability that all six of his ACKs failed is at most $(0.1)^6$, or one in a million. It is not necessary in this case that Bob actually keep count of Alice's retry attempts.

For a TCP example, see *17 TCP Transport Basics*, exercise 4.0.

TFTP addresses the lost-final-ACK problem by recommending (though not requiring) that the receiver enter into a **DALLY** state when it has sent the final ACK. In this state, the receiver responds only to duplicates of the final DATA packet; its response is to retransmit the final ACK. While one lost final ACK is possible, multiple such losses are unlikely; sooner or later the sender should receive the final ACK and will then exit.

The dally state will expire after an interval. This interval should be at least twice the sender's timeout interval, allowing for the sender to make three tries with the final data packet in all. Note that the receiver has no direct way to determine the sender's timeout value. Note also that dallying only provides increased assurance, not certainty: it is *possible* that all final ACKs were lost.

The TCP analogue of dallying is the TIMEWAIT state (*18.2 TIMEWAIT*), though TIMEWAIT also has another role related to prevention of old duplicates.

16.3.3 Duplicated Connection Request

We would also like to be able to distinguish between duplicated (*eg* retransmitted) connection requests and close but separate connection requests, especially when the second of two separate connection requests represents the cancellation of the first. Here is an outline in TFTP terms of the scenario we are trying to avoid:

- The client sends RRQ("foo")
- The client changes its mind, or aborts, or reboots, or whatever
- The client sends RRQ("bar")
- The server responds with Data[1] from the first RRQ, that is, with file "foo", while the client is expecting file "bar"

In correct TFTP operation, it is up to the client to send the second RRQ("bar") from a new port. As long as the client does that, changing its mind is not a problem. The server might end up sending Data[1] for file "foo" off into the void – that is, to the first client port – until it times out, as TFTP doesn't have a cancellation message exactly. But the request for file "bar" should succeed normally. One minor issue is that, when a TFTP application terminates, it may not have preserved anywhere a record of the port it used last, and so may be unable to guarantee that a new port is different from those used previously. But both strategies of *16.3.1 Old Duplicate Packets* – choosing a port number at random, and having the operating system assign one – are quite effective here.

TFTP does run into a somewhat unexpected issue, however, when the client sends a duplicate RRQ; typically this happens when the first RRQ times out. It is certainly possible to implement a TFTP server so as to recognize that the second RRQ is a duplicate, perhaps by noting that it is from the same client socket address and contains the same filename. In practice, however, this is incompatible with the simplified

implementation approach of *16.2 Trivial File Transport Protocol, TFTP* in which the server starts a new child process for each RRQ received.

What most TFTP server implementations do in this case is to start *two* sender processes, one for each RRQ received, from two ports s_port1 and s_port2. Both will send Data[1] to the receiver. The receiver is expected to "latch on" to the port of the first Data[1] packet it receives, recording its source port. The second Data[1] will now appear to be from an incorrect port. The TFTP specification requires that a receiver reply to any packets from an unknown port by sending an ERROR packet with the code "Unknown Transfer ID" (where "Transfer ID" means "port number"); this causes the sender process that sent the later-arriving Data[1] to shut down. The sender process that sent the winning Data[1] will continue normally. Were it not for this duplicate-RRQ scenario, packets from an unknown port could probably be simply ignored.

It is theoretically possible for a malicious actor on the LAN to take advantage of this TFTP "latching on" behavior to hijack anticipated RRQs. If the actor is aware that host C is about to request a file via TFTP, it might send repeated copies of bad Data[1] to likely ports on C. When C *does* request a file, it may receive the malicious file instead of what it asked for. Because the malicious application must guess the client's port number, though, this scenario appears to be of limited importance. However, many diskless devices do load a boot file on startup via TFTP, and may do so from a predictable port number.

16.3.4 Reboots

Any ongoing communications protocol has to take into account the possibility that one side may reboot in between messages from the other side. The primary issue is detection of the reboot, so the other side can close the now-broken connection.

If the sending side of a TFTP connection reboots, packet exchange simply stops, assuming a typical receiver that does not retransmit on timeouts. If the receiving side reboots, the sender will continue to send data packets, but will receive no further acknowledgments. In most cases, the newly rebooted client will simply ignore them.

The second issue with reboots is that the rebooting system typically loses all memory of what ports it has used recently, making it difficult to ensure that it doesn't reuse recently active ports. This leads to some risk of old duplicates.

Here is a scenario, based on the one at the start of the previous section, in which a client reboot leads to receipt of the wrong file. Suppose the client sends RRQ("foo"), but then reboots before sending ACK[1]. After reboot, the client then sends RRQ("bar"), from the same port; after the reboot the client will be unable to guarantee not reopening a recently used port. The server, having received the RRQ("foo"), belatedly proceeds to send Data[1] for "foo". The client latches on to this, and accepts file "foo" while believing it is receiving file "bar".

In practical terms, this scenario seems to be of limited importance, though "diskless" devices often do use TFTP to request their boot image file when restarting, and so might be potential candidates.

16.4 Other TFTP notes

We now take a brief look at other aspects of TFTP unrelated to the fundamental transport issues above. We include a brief outline of an implementation.

16.4.1 TFTP and the Sorcerer

TFTP uses a very straightforward implementation of stop-and-wait (*8.1 Building Reliable Transport: Stop-and-Wait*). Acknowledgment packets contain the block number of the data packet being acknowledged; that is, ACK[N] acknowledges Data[N].

In the original **RFC 783** specification, TFTP was vulnerable to the Sorcerer's Apprentice bug (*8.1.2 Sorcerer's Apprentice Bug*). Correcting this problem was the justification for updating the protocol in **RFC 1350**, eleven years later. The omnibus hosts-requirements document **RFC 1123** (referenced by **RFC 1350**) describes the necessary change this way:

> Implementations MUST contain the fix for this problem: the sender (*ie*, the side originating the DATA packets) must never resend the current DATA packet on receipt of a duplicate ACK.

16.4.2 TFTP States

The TFTP specification is relatively informal; more recent protocols are often described using finite-state terminology. In each allowable state, such a specification spells out the appropriate response to all packets. We can apply this approach to TFTP as well.

Above we defined a DALLY state, for the receiver only, with a specific response to arriving Data[N] packets. There are two other important conceptual states for TFTP receivers, which we might call UNLATCHED and ESTABLISHED.

When the receiver-client first sends RRQ, it does not know the port number from which the sender will send packets. We will call this state UNLATCHED, as the receiver has not "latched on" to the correct port. In this state, the receiver waits until it receives a packet from the sender that *looks like* a Data[1] packet; that is, it is from the sender's IP address, it has a plausible length, it is a DATA packet, and its block number is 1. When this packet is received, the receiver records s_port, and enters the ESTABLISHED state.

Once in the ESTABLISHED state, the receiver verifies for all packets that the source port number is s_port. If a packet arrives from some other port, the receiver sends back to its source an ERROR packet with "Unknown Transfer ID", but continues with the original transfer.

Here is an outline, in java, of what part of the TFTP receiver source code might look like; the code here handles the ESTABLISHED state. Somewhat atypically, the code here times out and retransmits ACK packets if no new data is received in the interval `TIMEOUT`; generally timeouts are implemented only at the TFTP sender side. Error processing is minimal, though error responses *are* sent in response to packets from the wrong port as described in the previous section. For most of the other error conditions checked for, there is no defined TFTP response.

The variables `state`, `sendtime`, `TIMEOUT`, `thePacket`, `theAddress`, `thePort`, `blocknum` and `expected_block` would need to have been previously declared and initialized; `sendtime` represents the time the most recent ACK response was sent. Several helper functions, such as `getTFTPOpcode()` and `write_the_data()`, would have to be defined. The remote port `thePort` would be initialized at the time of entry to the ESTABLISHED state; this is the port from which a packet must have been sent if it is to be considered valid. The loop here transitions to the DALLY state when a packet marking the end of the data has been received.

```
// TFTP code for ESTABLISHED state

while (state == ESTABLISHED) {
    // check elapsed time
    if (System.currentTimeMillis() > sendtime + TIMEOUT) {
        retransmit_most_recent_ACK()
        sendtime = System.currentTimeMillis()
    // receive the next packet
    try {
        s.receive(thePacket);
    }
    catch (SocketTimeoutException stoe) { continue; }   // try again
    catch (IOException ioe) { System.exit(1); }         // other errors

    if (thePacket.getAddress() != theAddress) continue;
    if (thePacket.getPort() != thePort) {
        send_error_packet(...);                 // Unknown Transfer ID; see␣
↪text
        continue;
    }
    if (thePacket.getLength() < TFTP_HDR_SIZE) continue;   // TFTP_HDR_SIZE =␣
↪4
    opcode = thePacket.getData().getTFTPOpcode()
    blocknum  = thePacket.getData().getTFTPBlock()
    if (opcode != DATA) continue;
    if (blocknum != expected_block) continue;
    write_the_data(...);
    expected_block ++;
    send_ACK(...);              // and save it too for possible retransmission
    sendtime = System.currentTimeMillis();
    datasize = thePacket.getLength() - TFTP_HDR_SIZE;
    if (datasize < MAX_DATA_SIZE) state = DALLY;  // MAX_DATA_SIZE = 512
}
```

Note that the check for elapsed time is quite separate from the check for the `SocketTimeoutException`. It is possible for the receiver to receive a steady stream of "wrong" packets, so that it never encounters a `SocketTimeoutException`, and yet no "good" packet arrives and so the receiver must still arrange (as above) for a timeout and retransmission.

16.4.3 TFTP Throughput

On a single physical Ethernet, the TFTP sender and receiver would alternate using the channel, with very little "turnaround" time; the effective throughput would be close to optimal.

As soon as the store-and-forward delays of switches and routers are introduced, though, stop-and-wait becomes a performance bottleneck. Suppose that the path from sender A to receiver B passes through two switches: A—S1—S2—B, and that on all three links only the bandwidth delay is significant. Because ACK packets are so much smaller than DATA packets, we can effectively ignore the ACK travel time from B to A.

With these assumptions, the throughput is about a third of the underlying bandwidth. This is because only

one of the three links can be active at any given time; the other two must be idle. We could improve throughput threefold by allowing A to send three packets at a time:

- packet 1 from A to S1
- packet 2 from A to S1 while packet 1 goes from S1 to S2
- packet 3 from A to S1 while packet 2 goes from S1 to S2 and packet 1 goes from S2 to B

This amounts to sliding windows with a winsize of three. TFTP does not support this; in the next chapter we study TCP, which does.

16.5 Remote Procedure Call (RPC)

A very different communications model, usually but not always implemented over UDP, is that of **Remote Procedure Call**, or RPC. The name comes from the idea that a procedure call is being made over the network; host A packages up a *request*, with parameters, and sends it to host B, which returns a *reply*. The term **request/reply protocol** is also used for this. The side making the request is known as the *client*, and the other side the *server*.

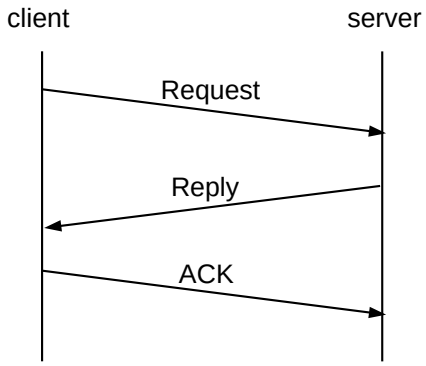
They're not quite procedure calls

The RPC name is not entirely a good fit. You can't pass pointers as parameters, for example, and true procedure calls seldom time out. See [TR88] for more. But the name has stuck.

One common example is that of DNS: a host sends a DNS lookup request to its DNS server, and receives a reply. Other examples include password verification, system information retrieval, database queries and file I/O (below). RPC is also quite successful as the mechanism for interprocess communication within CPU clusters, perhaps its most time-sensitive application.

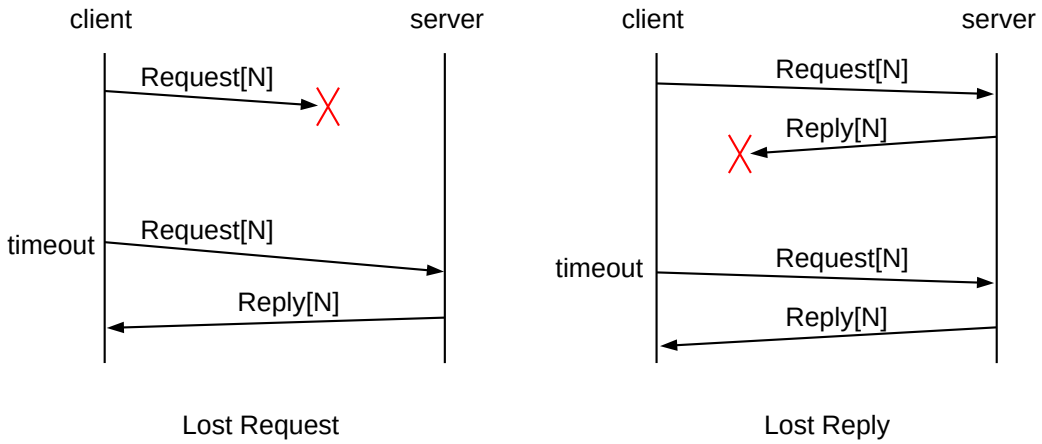
While TCP can be used for processes like these, this adds the overhead of creating and tearing down a connection; in many cases, the RPC exchange consists of nothing further beyond the request and reply and so the TCP overhead would be nontrivial. RPC over UDP is particularly well suited for transactions where both endpoints are quite likely on the same LAN, or are otherwise situated so that packet losses are negligible.

One issue with the use of UDP is that any desired acknowledgements have to be implemented within the RPC layer. This is not terribly difficult; usually the reply serves to acknowledge the request, and all that is needed is another ACK after that. If the protocol is run over a LAN, it is reasonable to use a static timeout period, perhaps somewhere in the range of 0.5 to 1.0 seconds. The diagram below includes an ACK.



Perhaps surprisingly, some RPC protocols omit the final ACK; see *16.5.2 Sun RPC* below. At a minimum, not having a final ACK means that if the reply is lost, the client has to start the sequence over, and the reply has to be regenerated from scratch.

It is essential that requests and replies be numbered (or otherwise identified), so that the client can determine which reply matches which request. This also means that the reply can serve to acknowledge the request; if reply[N] is not received; the requester retransmits request[N]. This can happen either if request[N] never arrived, or if it was reply[N] that got lost:



When the server creates reply[N] and sends it to the client, it must also keep a cached copy of the reply, until such time as ACK[N] is received.

After sending reply[N], the server may receive ACK[N], indicating all is well, or may receive request[N] again, indicating that reply[N] was lost, or may experience a timeout, indicating that either reply[N] or ACK[N] was lost. In the latter two cases, the server should retransmit reply[N] and wait again for ACK[N].

Finally, let us suppose that the server host delivers to its request-processing application the first copy of each request[N] to arrive, and that **neither side crashes** (or otherwise loses state in the middle of any one request/reply/ACK sequence). Let us also assume that no **packet reordering** occurs, and every request[N], reply[N] or ACK[N], retransmitted often enough, eventually makes it to its destination. We then have **exactly-once semantics**: while requests may be transmitted multiple times, they are processed (or "executed") once and only once.

16.5.1 Network File System

In terms of total packet volume, the application making the greatest use of early RPC was Sun's **Network File System**, or NFS; this allowed for a filesystem on the server to be made available to clients. When the client opened a file, the server would send back a *file handle* that typically included the file's identifying "inode" number. For `read()` operations, the request would contain the block number for the data to be read, and the corresponding reply would contain the data itself; blocks were generally 8 kB in size. For `write()` operations, the request would contain the block of data to be written together with the block number; the reply would contain an acknowledgment that it was received.

Usually an 8 kB block of data would be sent as a single UDP/IPv4 packet, using IPv4 fragmentation by the sender for transmission over Ethernet.

16.5.2 Sun RPC

The original simple model above is quite serviceable. However, in the RPC implementation developed by Sun Microsystems and documented in **RFC 1831** (and now officially known as Open Network Computing, or ONC, RPC), the final acknowledgment was omitted. As there are relatively few packet losses on a LAN, this was not quite as serious as it might sound, but it did have a major consequence: the server could now not afford to cache replies, as it would never receive an indication that it was ok to delete them. Therefore, the request was re-executed upon receipt of a second request[N], as in the right-hand "lost reply" diagram above.

This was often described as **at-least-once** semantics: if a client sent a request, and eventually received a reply, the client could be sure that the request was executed at least once, but if a reply got lost then the request might be transmitted more than once. Applications, therefore, had to be aware that this was a possibility.

It turned out that for many requests, duplicate execution of the reply was not a problem. A request that has the same result (and same side effects on the server) whether executed once or executed twice is known as **idempotent**. While a request to read or write the *next* block of a file is not idempotent, a request to read or write block 37 (or any other specific block) *is* idempotent. Most data queries are also idempotent; a second query simply returns the same data as the first. Even file `open()` operations are idempotent, or at least can be implemented as such: if a file is opened the second time, the file handle is simply returned a second time.

Alas, there do exist fundamentally non-idempotent operations. File locking is one, or any form of *exclusive* file open. Creating a directory is another, because the operation must fail if the directory already exists. Even opening a file is not idempotent if the server is expected to keep track of how many `open()` operations have been called, in order to determine if a file is still in use.

So why did Sun RPC take this route? One major advantage of at-least-once semantics is that it allowed the server to be **stateless**. The server would not need to maintain any RPC state, because without the final ACK there is no server RPC state to be maintained; for idempotent operations the server would generally not have to maintain any application state either. The practical consequence of this was that a server could crash and, because there was no state to be lost, could pick up right where it left off upon restarting.

Statelessness Inaction

Back when the Loyola CS department used Sun NFS extensively, server crashes would bring people calmly out of their offices to find out what had happened; client-workstation processes doing I/O would have locked up. Everyone would mill about in the hall until the server was rebooted, at which point they would return to their work and were almost always able to *pick up where they left off*. If the server had not been stateless, users would have been quite a bit less happy.

It is, of course, also possible to build recovery mechanisms into stateful protocols.

And for all that at-least-once semantics might sound like an egregious and obsolete shortcut, it does tend to be fast. Note, too, that the exactly-once protocol outlined in the final paragraph of *16.5 Remote Procedure Call (RPC)* includes the requirement that neither side crashes. A few approaches to the crash-and-reboot problem are reviewed in *16.5.4 RPC Refinements*.

The lack of file-locking and other non-idempotent I/O operations, along with the rise of cheap client-workstation storage (and, for that matter, more-reliable servers), eventually led to the decline of NFS over RPC, though it has not disappeared. NFS can, if desired, also be run (statefully!) over TCP.

Sun RPC also includes a data-encoding standard known as eXternal Data Representation, or **XDR**, eventually standardized in **RFC 1832**. It describes a way of encoding standard data types as sequences of bytes, ready for transmission. Integer values, for example, are encoded in big-endian format. Data transmitted via XDR is *not* tagged with its type, unlike, for example, the encoding of *26.12 SNMP and ASN.1 Encoding*. This means the sender and receiver have to agree on the precise parameter type signature for each RPC call. With Sun RPC this was typically managed with **rpcgen**, a tool which takes an XDR-compatible representation of the parameter types and generates code for parameter packing and unpacking.

16.5.3 Serial Execution

In some RPC systems, even those with explicit ACKs, requests are executed serially by the server. Serial execution is a necessity if request[N+1] serves as an implicit ACK[N]. Serial execution is a problem for file I/O operations, as physical disk drives are generally most efficient when the I/O operations can be reordered to suit the geometry of the disk. Disk drives commonly use the **elevator algorithm** to process requests: the read head moves from low-numbered tracks outwards to high-numbered tracks, pausing at each track for which there is an I/O request. Waiting for the Nth read to complete before asking the disk to start the N+1th one is slow.

The best solution here, from the server application's perspective, is to allow multiple outstanding requests and out-of-order replies. This complicates the RPC protocol, however.

16.5.4 RPC Refinements

One basic network-level improvement to RPC concerns the avoidance of IP-level fragmentation. While fragmentation is not a major performance problem on a single LAN, it may have difficulties over longer distances. One possible refinement is an RPC-level large-message protocol, that fragments at the RPC layer and which supports a mechanism for retransmission, if necessary, only of those fragments that are actually lost.

Another optimization might address the possibility that the client or the server might crash and reboot. To detect client restarts we can add to the client side a "boot counter", incremented on each reboot and then

rewritten to persistent storage. This value is then included in each request, and echoed back in each reply and ACK. This allows the server to distinguish between requests sent before and after a client reboot; such requests are conceptually unrelated and the mechanism here ensures they receive different identifiers. See *27.3.3 SNMPv3 Engines* for a related example.

On the server side, allowing for crashes and reboots is even more complicated. If the goal is simply to make the client aware that the server may have rebooted during a request/reply sequence, we might include the server's boot counter in each reply[N]; if the client sees a change, there may be a problem with the current request. We might also include the client's current estimate of the server's boot counter in each request, and have the server deny requests for which there is a mismatch.

In exceptional cases, we can liken requests to database transactions and include on the server side a database-style crash-recovery journal. The goal is to allow the server, upon restarting, to identify requests that were in progress at the time of the crash, and either to roll them back or to complete them. This is not trivial, and can only be done for restricted classes of requests (*eg* reads and writes).

16.5.5 gRPC

The "g" here is for Google. gRPC was designed for request/reply operations where the requests or replies may be large and complex, or where the client and the server are *not* on the same LAN; it is well suited for end-user requests to large servers. The underlying transport is TCP. More specifically, data is sent each way using HTTP/2, which has support for multiple data streams (though eventually gRPC seems likely to migrate to QUIC-based HTTP/3, which uses UDP (*16.1.1 QUIC*)). The use of TLS (*29.5.2 TLS*) is also supported, for authentication and encryption; these are essential for long-haul connections but are less so within a datacenter. gRPC is also well-suited for cases – even within a datacenter – where requests are **not** idempotent and where lost responses could be serious.

gRPC in effect focuses on the encoding portion of RPC; this is the part of SunRPC handled by XDR. It supports streamed data, though, which XDR does not.

16.5.6 Homa

Homa ([MLAO18]), on the other hand, is *not* meant for long-distance communications. It is intended to be a very high-performance RPC implementation for use exclusively within datacenters, generally where requests and replies are relatively small. The primary design goal is the minimization of latency. At 10 Gbps, one full-sized packet can be transmitted in about 1.2 µsec, and the implementation of [MLAO18] achieves, at 80% network load, 15 µsec delivery times for 99% of requests.

Fast datacenter RPC is a very active research area, and Homa has quite a few predecessors. Among these are pHost ([GNKARS15]), pFabric ([AYSKMP13], and which requires special switches), and FastPass ([POBSF14], which requires a central scheduler).

Queuing delay is the largest delay culprit here, and Homa addresses this by explicitly setting the Ethernet VLAN priority field for packets (*3.2 Virtual LAN (VLAN)*). There are eight priority levels available. Homa adjusts packet priorities with the goal of giving the highest priority to responses that have the fewest remaining packets; this is known as Shortest Remaining Processing Time (SRPT) first.

Senders of data responses send the first chunk of data (typically up to 10 KB) "blindly"; that is, without a receiver-supplied priority. Blind transmission does not mean default priority, though; receivers continually

monitor current traffic conditions and piggyback their recommendations for blind-traffic priority on other Homa traffic. This first data chunk also includes information about the total size of the data. After sending the first chunk, a sender waits for a "GRANT" message from the receiver, which includes the receiver's chosen priorities for the remaining data. The receiver sets the priorities because, in a typical datacenter, queues form primarily at the so-called "top of rack" switches nearest to the receiver, and so it is the receiver that is best positioned to manage these queues.

Homa's use of priorities – both blind and GRANT – are what allow Homa data to leapfrog larger, non-Homa data flows. The use of priorities also largely prevents the "incast" congestion problem (see *22.13.1 TCP Incast*) when a host sends out multiple requests and receives the corresponding replies all at the same time. For Sun NFS with 8 KB data blocks, blind priorities would be used frequently, though GRANT priorities would still come into play for multi-block messages. For message sizes in the range 10 KB - 1 MB, *eg* the Hadoop example of [MLAO18], most data would be transmitted under the aegis of GRANT priorities. In accordance with the SRPT strategy, the priorities for the packets of a large message would steadily increase as the message was transmitted.

Homa, like SunRPC, does not support final acknowledgments of data; if a request is made and *all* the response data packets are lost, then when the request is retransmitted the response will be evaluated again from scratch. This results in "at-least-once" semantics, but significantly simplifies the overall protocol and tends thereby to improve throughput. However, if individual packets are lost, the receiver sends a RESEND message for the missing byte range. The sender will most likely not have cached the data, because, as with SunRPC, without a final ACK it cannot know when to delete the cached response, and so the response will again be evaluated from scratch.

In sending GRANT requests, receivers engage in carefully calculated "overcommitment"; that is, receivers grant more data transmissions than can be delivered immediately without queuing. This is because senders are not always able to send more data immediately, typically because they may also be in the process of sending other data to other receivers.

16.6 Epilog

UDP does not get as much attention as TCP, but between avoidance of connection-setup overhead, avoidance of head-of-line blocking and high LAN performance, it holds its own.

We also use UDP here to illustrate fundamental transport issues, both abstractly and for the specific protocol TFTP. We will revisit these fundamental issues extensively in the next chapter in the context of TCP; these issues played a major role in TCP's design.

16.7 Exercises

Exercises may be given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercises marked with a ♢ have solutions or hints at *34.13 Solutions for UDP*.

1.0. Perform the UDP simplex-talk experiments discussed at the end of *16.1.3 UDP Simplex-Talk*. Can multiple clients have simultaneous sessions with the same server?

2.0. Suppose that both sides of a TFTP transfer implement retransmit-on-timeout and neither side implements retransmit-on-duplicate. What would happen in each of the following cases if the first Data[3] packet

is lost?

(a)♢. Sender timeout = receiver timeout = 2 seconds.
(b). Sender timeout = 1 second; receiver timeout = 3 seconds.
(c). Sender timeout = 3 seconds; receiver timeout = 1 second.

Assume the actual transfer time is negligible in comparison to the timeout intervals, and that the retransmitted Data[3] is received successfully.

3.0. In the previous exercise, how do things change if the first ACK[3] is the packet that is lost?

4.0. For each state below, spell out plausible responses for a TFTP receiver upon receipt of a Data[N] packet. Your answers may depend on N and the packet size. Indicate the events that cause a transition from one state to the next. The TFTP states were proposed in *16.4.2 TFTP States*.

(a). UNLATCHED
(b). ESTABLISHED
(c). DALLYING

Example: upon receipt of an ERROR packet, TFTP would in all three states exit.

5.0. In the TFTP-receiver code in *16.4.2 TFTP States*, explain why we must check `thePacket.getLength()` *before* extracting the opcode and block number.

6.0. Assume both the TFTP sender and the TFTP receiver implement retransmit-on-timeout but *not* retransmit-on-duplicate. Outline a specific TFTP scenario in which the TFTP receiver of *16.4.2 TFTP States* sets a socket timeout interval but never encounters a "hard" timeout – that is, a `SocketTimeoutException` – and yet must timeout and retransmit. Hint: arrange so the sender regularly times out and retransmits some packet, at an interval less than the receiver's `SocketTimeoutException` time, but it is not the packet the receiver is waiting for.

7.0. At the end of *16.3.1 Old Duplicate Packets*, we claimed that if either side in the TFTP protocol changed ports, the old-duplicate problem would not occur.

(a). If the client (receiver) changes its port number on a subsequent connection, but the server (sender) does not, what prevents an old-duplicate data packet sent by the server from being accepted by the new client?
(b). If the server changes its port number on a subsequent connection, but the client does not, what prevents an old-duplicate DATA[N] packet, with N>1, sent by the server from being accepted by the new client?

8.0. In part (b) of the previous exercise, it was claimed that an old-duplicate DATA[N] could not be accepted as valid by the new receiver provided N>1. Give an example in which an old-duplicate DATA[1] *is* accepted as valid.

9.0. Suppose a TFTP server implementation resends DATA[N] on receipt of a duplicate ACK[N-1], contrary to *16.4.1 TFTP and the Sorcerer*. It receives a file request from a *partially implemented* TFTP client, that sends ACK[1] to the correct new port but then never increments the ACK number; the client's response to DATA[N] is always ACK[1]. What happens? (Based on a true story.)

10.0. In the simple RPC protocol at the beginning of *16.5 Remote Procedure Call (RPC)*, suppose that the server sends reply[N] and experiences a timeout, receiving nothing back from the client. In the text we suggested that most likely this meant ACK[N] was lost. Give another loss scenario, involving the loss of two packets. Assume the client and the server have the same timeout interval.

11.0. Suppose a Sun RPC `read()` request ends up executing twice. Unfortunately, in between successive `read()` operations the block of data is updated by another process, so different data is returned. Is this a failure of idempotence? Why or why not?

12.0. Outline an RPC protocol in which multiple requests can be outstanding, and replies can be sent in any order. Assume that requests are numbered, and that ACK[N] acknowledges reply[N]. Should ACKs be cumulative? If not, what should happen if an ACK is lost?

13.0. Consider the request[N]/reply[N]/ACK[N] protocol of *16.5 Remote Procedure Call (RPC)*, under the assumption that requests are numbered sequentially, but packets may potentially be delivered out of order. Thus, request[5] may arrive again after ACK[5] has been sent. and the first request[5] may even arrive after ACK[6] has been sent.

(a). If requests are handled serially, as in *16.5.3 Serial Execution*, what information does the server side need to maintain in order to avoid duplicate execution of a request?

(b). What information does the server side need to maintain if requests are handled non-serially, that is, there can be multiple outstanding requests at any one time? Assume that if request[6] arrives before request[5], the server responds with reply[6] even though there is now a temporary gap in sequence numbers.

14.0. Suppose an RPC client maintains a boot counter as in *16.5.4 RPC Refinements*. Draw diagrams for cases (a) and (b), and indicate how the boot counter is used to resolve the situation.

(a). The client sends request[N], but reboots before reply[N] is received.

(b). The client sends request[N], and then immediately reboots and sends an unrelated request that just happens also to be numbered N.

(c). What would happen in the scenario in part (b) if the reply[N] packet did *not* echo back the boot-counter value from the request[N] packet?

15.0. In this exercise we explore UDP connection state using `netcat` (*16.1.4 netcat*). Let A and B be two hosts (not necessarily distinct!).

(a). Verify that you can exchange messages between A and B after starting the following; `-u` is for UDP and `-l` is to create the server side (to "listen").

In a terminal on B: `netcat -u -l 5432`
In a terminal on A: `netcat -u B 5432`

(b). Now kill the `netcat` on A and restart it. A different local port is likely chosen by the second `netcat`; verify that communication fails.

(c). Now repeat the process, but this time in addition specify the *source* port on A with the `-p` option:
In a terminal on B: `netcat -u -l 5432`
In a terminal on A: `netcat -u -p 2345 B 5432`
Verify that killing and restarting the client on A allows communication to continue.

16.0. In this exercise we explore sending UDP packets through NAT routers (*9.7 Network Address Translation*), using `netcat` (*16.1.4 netcat*). Let A be an internal host, NR the *public* IP address of the NAT router, and C an outside host. We will initiate all connections by having A send to C at port 5432, which must not be firewalled (changing to a different port is straightforward).

(a). Verify that you can send from A to C:
In a terminal on C: `netcat -u -l 5432`
In a terminal on A: `netcat -u C 5432`

If this does not work, try changing port numbers or C's firewall settings.

(b). Try typing text into the terminal on C; `netcat` supports bidirectional communication. Does the output appear on A?

(c). Through experimentation, estimate the allowable delay between the A-to-C packets and the C-to-A response. 1 minute? 5 minutes? 10 minutes?

(d). Try to transmit the reply from C using an entirely separate pair of `netcat` sessions. For this to have any chance of working, A's source port must be known; we will set it here to 40001.
On C: as above
On A: `netcat -u -p 40001 C 5432`

As soon as data has been transmitted successfully from A to C, try the reverse path. Both A-to-C netcat processes, above, must first be terminated, to free the ports. Then:
On A: `netcat -u -l 40001`
On C: `netcat -u -p 5432 NR 40001`

17 TCP TRANSPORT BASICS

The standard transport protocols riding above the IP layer are **TCP** and **UDP**. As we saw in *16 UDP Transport*, UDP provides simple datagram delivery to remote sockets, that is, to ⟨host,port⟩ pairs. TCP provides a much richer functionality for sending data to (connected) sockets. In this chapter we cover the basic TCP protocol; in the following chapter we cover some subtle issues related to potential data loss, some TCP implementation details, and then some protocols that serve as alternatives to TCP.

TCP is quite different in several dimensions from UDP. TCP is **stream-oriented**, meaning that the application can write data in very small or very large amounts and the TCP layer will take care of appropriate packetization (and also that TCP transmits a stream of bytes, not messages or records; *cf 18.15.2 SCTP*). TCP is **connection-oriented**, meaning that a connection must be established before the beginning of any data transfer. TCP is **reliable**, in that TCP uses sequence numbers to ensure the correct order of delivery and a timeout/retransmission mechanism to make sure no data is lost short of massive network failure. Finally, TCP automatically uses the **sliding windows** algorithm to achieve throughput relatively close to the maximum available.

These features mean that TCP is very well suited for the transfer of large files. The two endpoints open a connection, the file data is written by one end into the connection and read by the other end, and the features above ensure that the file will be received correctly. TCP also works quite well for interactive applications where each side is sending and receiving streams of small packets. Examples of this include ssh or telnet, where packets are exchanged on each keystroke, and database connections that may carry many queries per second. TCP even works *reasonably* well for **request/reply** protocols, where one side sends a single message, the other side responds, and the connection is closed. The drawback here, however, is the overhead of setting up a new connection for each request; a better application-protocol design might be to allow multiple request/reply pairs over a single TCP connection.

Note that the connection-orientation and reliability of TCP represent abstract features built on top of the IP layer, which supports neither of them.

The connection-oriented nature of TCP warrants further explanation. With UDP, if a server opens a socket (the OS object, with corresponding socket address), then any client on the Internet can send to that socket, via its socket address. Any UDP application, therefore, must be prepared to check the source address of each packet that arrives. With TCP, all data arriving at a *connected* socket must come from the other endpoint of the connection. When a server S initially opens a socket s, that socket is "unconnected"; it is said to be in the LISTEN state. While it still has a socket address consisting of its host and port, a LISTENing socket will never receive data directly. If a client C somewhere on the Internet wishes to send data to s, it must first establish a connection, which will be defined by the **socketpair** consisting of the socket addresses (that is, the ⟨IP_addr,port⟩ pairs) at both C and S. As part of this connection process, a new *connected* child socket s_C will be created; it is s_C that will receive any data sent from C. Usually, the server will also create a new thread or process to handle communication with s_C. Typically the server will have multiple connected children of the original socket s, and, for each one, a process attached to it.

If C1 and C2 both connect to s, two connected sockets at S will be created, s_1 and s_2, and likely two separate processes. When a packet arrives at S addressed to the socket address of s, the *source* socket address will also be examined to determine whether the data is part of the C1–S or the C2–S connection, and thus whether a read on s_1 or on s_2, respectively, will see the data.

If S is acting as an ssh server, the LISTENing socket listens on port 22, and the connected child sockets correspond to the separate user login connections; the process on each child socket represents the login process of that user, and may run for hours or days.

In Chapter 1 we likened TCP sockets to telephone connections, with the server like one high-volume phone number 800-BUY-NOWW. The unconnected socket corresponds to the number everyone dials; the connected sockets correspond to the actual calls. (This analogy breaks down, however, if one looks closely at the way such multi-operator phone lines are actually configured: each typically *does* have its own number.)

TCP was originally defined in **RFC 793**, with important updates in **RFC 1122**, dated October 1989. Since then there have been many miscellaneous updates; all of these have now been incorporated into a single specification **RFC 9293**.

17.1 The End-to-End Principle

The End-to-End Principle is spelled out in [SRC84]; it states in effect that transport issues are the responsibility of the endpoints in question and thus should not be delegated to the core network. This idea has been very influential in TCP design.

Two issues falling under this category are data corruption and congestion. For the first, even though essentially all links on the Internet have link-layer checksums to protect against data corruption, TCP still adds its own checksum (in part because of a history of data errors introduced *within* routers). For the latter, TCP is today essentially the *only* layer that addresses congestion management.

Saltzer, Reed and Clark categorized functions that were subject to the End-to-End principle this way:

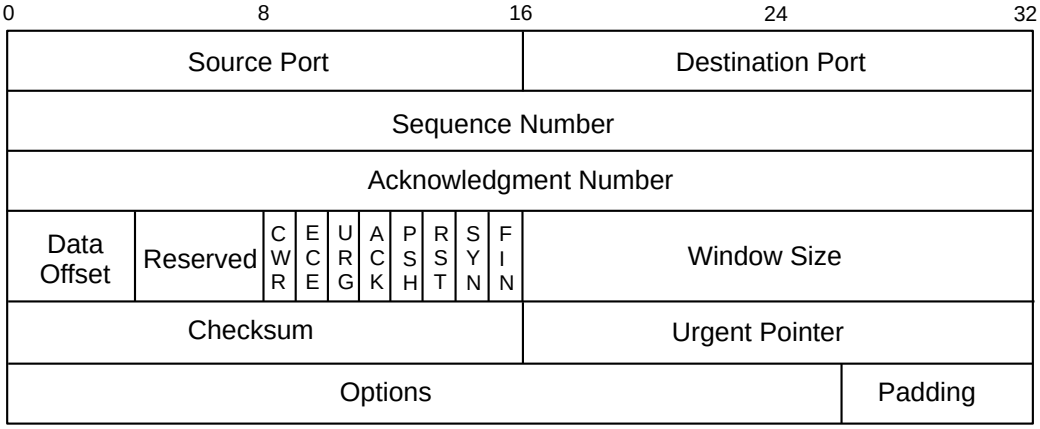
> The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)

This does not mean that the backbone Internet should not concern itself with congestion; it means that backbone congestion-management mechanisms should not completely replace end-to-end congestion management.

17.2 TCP Header

Below is a diagram of the TCP header. As with UDP, source and destination ports are 16 bits. The 4-bit Data Offset field specifies the number of 32-bit words in the header; if no options are present its value is 5.

As with UDP, the checksum covers the TCP header, the TCP data and an IP "pseudo header" that includes the source and destination IP addresses. The checksum must be updated by a NAT router that modifies any header values. (Although the IP and TCP layers are theoretically separate, and **RFC 793** in some places appears to suggest that TCP can be run over a non-IP internetwork layer, **RFC 793** also explicitly defines 4-byte addresses for the pseudo header. **RFC 2460** officially redefined the pseudo header to allow IPv6 addresses.)



The **sequence** and **acknowledgment** numbers are for numbering the data, at the byte level. This allows TCP to send 1024-byte blocks of data, incrementing the sequence number by 1024 between successive packets, or to send 1-byte telnet packets, incrementing the sequence number by 1 each time. There is no distinction between DATA and ACK packets; all packets carrying data from A to B also carry the most current acknowledgment of data sent from B to A. Many TCP applications are largely unidirectional, in which case the sender would include essentially the same acknowledgment number in each packet while the receiver would include essentially the same sequence number.

It is traditional to refer to the data portion of TCP packets as **segments**.

TCP History

The clear-cut division between the IP and TCP headers did not spring forth fully formed. See [CK74] for a discussion of a proto-TCP in which the sequence number (but not the acknowledgment number) appeared in the equivalent of the IP header (perhaps so it could be used for fragment reassembly).

The value of the sequence number, in *relative* terms, is the position of the first byte of the packet in the data stream, or the position of what would be the first byte in the case that no data was sent. The value of the acknowledgment number, again in relative terms, represents the byte position for the next byte expected. Thus, if a packet contains 1024 bytes of data and the first byte is number 1, then that would be the sequence number. The data bytes would be positions 1-1024, and the ACK returned would have acknowledgment number 1025.

The sequence and acknowledgment numbers, as sent, represent these relative values *plus* an **Initial Sequence Number**, or ISN, that is fixed for the lifetime of the connection. Each direction of a connection has its own ISN; see below.

TCP acknowledgments are **cumulative**: when an endpoint sends a packet with an acknowledgment number of N, it is acknowledging receipt of all data bytes numbered less than N. Standard TCP provides no mechanism for acknowledging receipt of packets 1, 2, 3 and 5; the highest cumulative acknowledgment that could be sent in that situation would be to acknowledge packet 3.

The TCP header defines some important flag bits; the brief definitions here are expanded upon in the sequel:

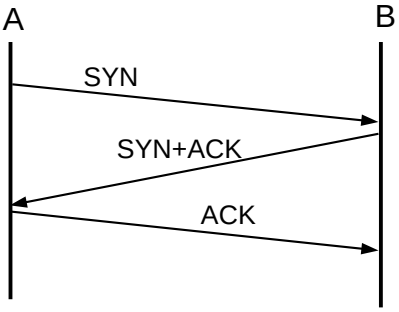
- **SYN**: for SYNchronize; marks packets that are part of the new-connection handshake

- **ACK**: indicates that the header Acknowledgment field is valid; that is, all but the first packet
- **FIN**: for FINish; marks packets involved in the connection closing
- **PSH**: for PuSH; marks "non-full" packets that should be delivered promptly at the far end
- **RST**: for ReSeT; indicates various error conditions
- **URG**: for URGent; part of a now-seldom-used mechanism for high-priority data
- **CWR** and **ECE**: part of the Explicit Congestion Notification mechanism, *21.5.3 Explicit Congestion Notification (ECN)*

17.3 TCP Connection Establishment

TCP connections are established via an exchange known as the **three-way handshake**. If A is the client and B is the LISTENing server, then the handshake proceeds as follows:

- A sends B a packet with the SYN bit set (a SYN packet)
- B responds with a SYN packet of its own; the ACK bit is now also set
- A responds to B's SYN with its own ACK



TCP three-way handshake

Normally, the three-way handshake is triggered by an application's request to connect; data can be sent only after the handshake completes. This means a one-RTT delay before any data can be sent. The original TCP standard **RFC 793** does allow data to be sent with the first SYN packet, as part of the handshake, but such data cannot be released to the remote-endpoint application until the handshake completes. Most traditional TCP programming interfaces offer no support for this early-data option.

There are recurrent calls for TCP to support data transmission within the handshake itself, so as to achieve request/reply turnaround comparable to that with RPC (*16.5 Remote Procedure Call (RPC)*). We return to this in *18.5 TCP Faster Opening*.

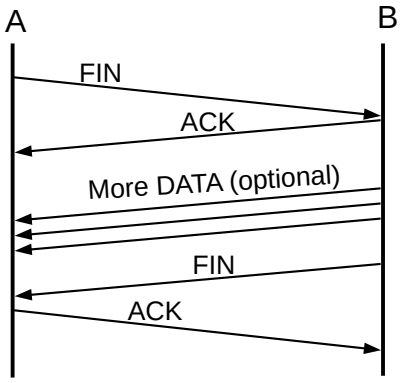
The three-way handshake is vulnerable to an attack known as **SYN flooding**. The attacker sends a large number of SYN packets to a server B. For each arriving SYN, B must allocate resources to keep track of what appears to be a legitimate connection request; with enough requests, B's resources may face exhaustion. SYN flooding is easiest if the SYN packets are simply spoofed, with forged, untraceable source-IP addresses; see spoofing at *9.1 The IPv4 Header*, and *18.3.1 ISNs and spoofing* below. SYN-flood attacks can also take the form of a large number of real connection attempts from a large number of real clients – often compromised and pressed into service by some earlier attack – but this requires considerably more resources

on the part of the attacker. See *18.15.2 SCTP* for an alternative handshake protocol (unfortunately not available to TCP) intended to mitigate SYN-flood attacks, at least from spoofed SYNs.

To **close** the connection, a superficially similar exchange involving FIN packets may occur:

- A sends B a packet with the FIN bit set (a FIN packet), announcing that it has finished sending data
- B sends A an ACK of the FIN
- B may continue to send additional data to A
- When B is also ready to cease sending, it sends its own FIN to A
- A sends B an ACK of the FIN; this is the final packet in the exchange

Here's the ladder diagram for this:



A typical TCP close

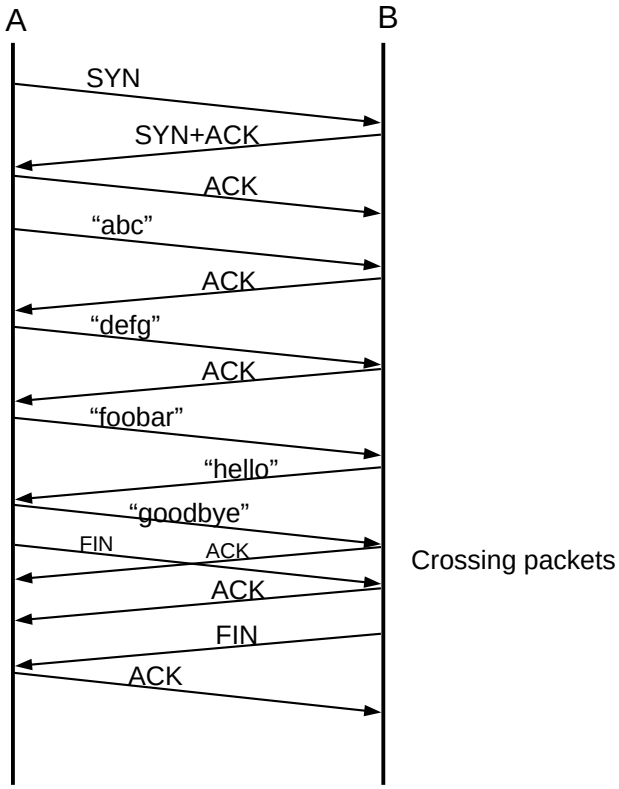
The FIN handshake is really more like two separate two-way FIN/ACK handshakes. We will return to TCP connection closing in *17.8.1 Closing a connection*.

Now let us look at the full exchange of packets in a representative connection, in which A sends strings "abc", "defg", and "foobar" (**RFC 3092**). B replies with "hello", and which point A sends "goodbye" and closes the connection. In the following table, *relative* sequence numbers are used, which is to say that sequence numbers begin with 0 on each side. The SEQ numbers in **bold** on the A side correspond to the ACK numbers in **bold** on the B side; they both count data flowing from A to B.

	A sends	B sends
1	SYN, **seq=0**	
2		SYN+ACK, seq=0, **ack=1** (expecting)
3	ACK, **seq=1**, ack=1 (ACK of SYN)	
4	“abc”, **seq=1**, ack=1	
5		ACK, seq=1, **ack=4**
6	“defg”, **seq=4**, ack=1	
7		seq=1, **ack=8**
8	“foobar”, **seq=8**, ack=1	
9		seq=1, **ack=14**, “hello”
10	**seq=14**, ack=6, “goodbye”	
11,12	**seq=21**, ack=6, FIN	seq=6, **ack=21** ;; ACK of “goodbye”, crossing packets
13		seq=6, **ack=22** ;; ACK of FIN
14		seq=6, **ack=22**, FIN
15	**seq=22**, ack=7 ;; ACK of FIN	

(We will see below that this table is slightly idealized, in that real sequence numbers do *not* start at 0.)

Here is the ladder diagram corresponding to this connection:



In terms of the sequence and acknowledgment numbers, SYNs count as 1 byte, as do FINs. Thus, the SYN counts as sequence number 0, and the first byte of data (the “a” of “abc”) counts as sequence number 1. Similarly, the ack=21 sent by the B side is the acknowledgment of “goodbye”, while the ack=22 is the

acknowledgment of A's subsequent FIN.

Whenever B sends ACN=n, A follows by sending more data with SEQ=n.

TCP does *not* in fact transport relative sequence numbers, that is, sequence numbers as transmitted do not begin at 0. Instead, each side chooses its **Initial Sequence Number**, or **ISN**, and sends that in its initial SYN. The third ACK of the three-way handshake is an acknowledgment that the server side's SYN response was received correctly. All further sequence numbers sent are the ISN chosen by that side plus the relative sequence number (that is, the sequence number as if numbering did begin at 0). If A chose ISN_A=1000, we would add 1000 to all the bold entries above: A would send SYN(seq=1000), B would reply with ISN_B and ack=1001, and the last two lines would involve ack=1022 and seq=1022 respectively. Similarly, if B chose ISN_B=7000, then we would add 7000 to all the **seq** values in the "B sends" column and all the **ack** values in the "A sends" column. The table above up to the point B sends "goodbye", with actual sequence numbers instead of relative sequence numbers, is below:

	A, ISN=**1000**	B, ISN=7000
1	SYN, **seq=1000**	
2		SYN+ACK, seq=7000, **ack=1001**
3	ACK, **seq=1001**, ack=7001	
4	"abc", **seq=1001**, ack=7001	
5		ACK, seq=7001, **ack=1004**
6	"defg", **seq=1004**, ack=7001	
7		seq=7001, **ack=1008**
8	"foobar", **seq=1008**, ack=7001	
9		seq=7001, **ack=1014**, "hello"
10	**seq=1014**, ack=7006, "goodbye"	

If B had not been LISTENing at the port to which A sent its SYN, its response would have been **RST** ("reset"), meaning in this context "connection refused". Similarly, if A sent data to B before the SYN packet, the response would have been RST.

Finally, a RST can be sent by either side at any time to abort the connection. Sometimes routers along the path send "spoofed" RSTs to tear down TCP connections they are configured to regard as undesired; see *9.7.2 Middleboxes* and **RFC 3360**. Worse, sometimes external attackers are able to tear down a TCP connection with a spoofed RST; this requires brute-force guessing the endpoint port numbers and the RST sender's current SEQ value (**RFC 793** does not in general require the RST packet's ACK value to match, but see exercise 9.0). In the days of 4 kB window sizes, guessing a valid SEQ value was a one-in-a-million chance, but window sizes have steadily increased (*21.6 The High-Bandwidth TCP Problem*); a 4 MB window size makes SEQ guessing quite feasible. See also **RFC 4953** and the RST-validation fix proposed in **RFC 5961** §3.2.

If A sends a series of small packets to B, then B has the option of assembling them into a full-sized I/O buffer before releasing them to the receiving application. However, if A sets the **PSH** bit on each packet, then B should release each packet immediately to the receiving application. In Berkeley Unix and most (if not all) BSD-derived socket-library implementations, there is in fact no way to set the PSH bit; it is set automatically for each write. (But even this is not *guaranteed* as the sender may leave the bit off or consolidate several PuSHed writes into one packet; this makes using the PSH bit as a record separator difficult. In a series of runs of the program written to generate the WireShark packet trace, below, most of the time the strings "abc", "defg", *etc* were PuSHed separately but occasionally they were consolidated into one packet.)

As for the **URG** bit, imagine a telnet (or ssh) connection, in which A has sent a large amount of data to B, which is momentarily stalled processing it. The application at A wishes to abort that processing by sending the interrupt character CNTL-C. Under normal conditions, the application at B would have to finish processing all the pending data before getting to the CNTL-C; however, the use of the URG bit can enable immediate asynchronous delivery of the CNTL-C. The bit is set, and the TCP header's Urgent Pointer field points to the CNTL-C in the current packet, far ahead in the normal data stream. The receiving application then skips ahead in its processing of the arriving data stream until it reaches the urgent data. For this to work, the receiving application process must have signed up to receive an asynchronous signal when urgent data arrives.

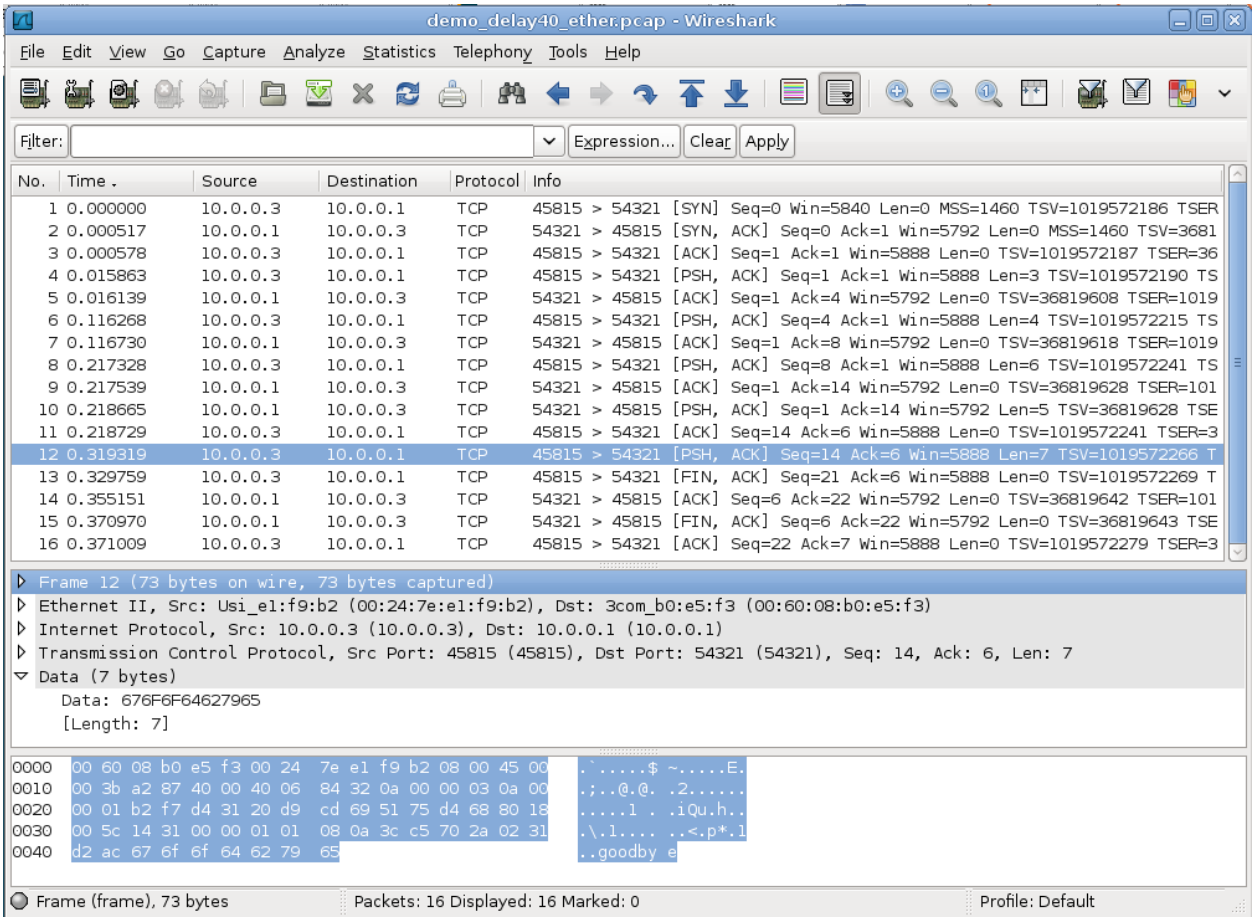
The urgent data does appear as part of the ordinary TCP data stream, and it is up to the protocol to determine the start of the data that is to be considered urgent, and what to do with the unread, buffered data sent ahead of the urgent data. For the CNTL-C example in the telnet protocol (**RFC 854**), the urgent data might consist of the telnet "Interrupt Process" byte, preceded by the "Interpret as Command" escape byte, and the earlier data is simply discarded.

Officially, the Urgent Pointer value, when the **URG** bit is set, contains the offset from the start of the current packet data to the *end* of the urgent data; it is meant to tell the receiver "you should read up to this point as soon as you can". The original intent was for the urgent pointer to mark the last byte of the urgent data, but §3.1 of **RFC 793** got this wrong and declared that it pointed to the first byte *following* the urgent data. This was corrected in **RFC 1122**, but most implementations to this day abide by the "incorrect" interpretation. **RFC 6093** discusses this and proposes, first, that the near-universal "incorrect" interpretation be accepted as standard, and, second, that developers avoid the use of the TCP urgent-data feature.

17.4 TCP and WireShark

Below is a screenshot of the WireShark program displaying a tcpdump capture intended to represent the TCP exchange above. Both hosts involved in the packet exchange were Linux systems. Side A uses socket address ⟨10.0.0.3,45815⟩ and side B (the server) uses ⟨10.0.0.1,54321⟩.

WireShark is displaying *relative* TCP sequence numbers. The first three packets correspond to the three-way handshake, and packet 4 is the first data packet. Every data packet has the flags [PSH, ACK] displayed. The data in the packet can be inferred from the WireShark Len field, as each of the data strings sent has a different length.



The packets are numbered the same as in the table above up through packet 8, containing the string "foobar". At that point the table shows B replying by a combined ACK plus the string "hello"; in fact, TCP sent the ACK alone and then the string "hello"; these are WireShark packets 9 and 10 (note packet 10 has Len=5). Wireshark packet 11 is then a standalone ACK from A to B, acknowledging the "hello". WireShark packet 12 (the packet highlighted) then corresponds to table packet 10, and contains "goodbye" (Len=7); this string can be seen at the right side of the bottom pane.

The table view shows A's FIN (packet 11) crossing with B's ACK of "goodbye" (packet 12). In the WireShark view, A's FIN is packet 13, and is sent about 0.01 seconds after "goodbye"; B then ACKs them both with packet 14. That is, the table-view packet 12 does not exist in the WireShark view.

Packets 11, 13, 14 and 15 in the table and 13, 14, 15 and 16 in the WireShark screen dump correspond to the connection closing. The program that generated the exchange at B's side had to include a "sleep" delay of 40 ms between detecting the closed connection (that is, reading A's FIN) and closing its own connection (and sending its own FIN); otherwise the ACK of A's FIN traveled in the same packet with B's FIN.

The ISN for A in this example was 551144795 and B's ISN was 1366676578. The actual pcap packet-capture file is at demo_tcp_connection.pcap. This pcap file was generated by a TCP connection between two physical machines; for an alternative approach to observing TCP behavior see *30.2.2 Mininet WireShark Demos*.

17.5 TCP Offloading

In the Wireshark example above, the hardware involved used **TCP checksum offloading**, or TCO, to have the network-interface card do the actual checksum calculations; this permits a modest amount of parallelism. As a result, the checksums for outbound packets are wrong in the capture file. WireShark has an option to disable the reporting of this. Despite the name, TCO can handle UDP packets as well.

Most Ethernet (and some Wi-Fi) cards have the ability to calculate the Internet checksum (*7.4 Error Detection*) over a certain range of bytes, and store the result (after taking the complement) at a designated offset. However, cards cannot, as a rule, handle the UDP and TCP "pseudo headers". So what happens is the host system calculates the pseudo-header checksum and stores it in the normal checksum field; the LAN card then includes this pseudo-header checksum value in its own checksum calculation, and the correct result is obtained.

It is also possible, with many newer network-interface cards, to offload the TCP *segmentation* process to the LAN hardware; that is, the kernel sends a very large TCP buffer – perhaps 64 KB – to the LAN hardware, along with a TCP header, and the LAN hardware divides the buffer into multiple TCP packets of at most 1500 bytes each. This is most useful when the application is writing data continuously and is known as **TCP segmentation offloading**, or TSO. The use of TSO requires TCO, but not vice-versa.

TSO can be divided into large *send* offloading, LSO, for outbound traffic, as above, and large receive offloading, LRO, for inbound. For inbound offloading, the network card accumulates multiple inbound packets that are part of the same TCP connection, and consolidates them in proper sequence to one much larger packet. This means that the network card, upon receiving one packet, must wait to see if there will be more. This wait is very short, however, at most a few milliseconds. Specifically, all consolidated incoming packets must have the same TCP Timestamp value (*18.4 Anomalous TCP scenarios*).

TSO is of particular importance at very high bandwidths. At 10 Gbps, a system can send or receive close to a million packets per second, and offloading some of the packet processing to the network card can be essential to maintaining high throughput. TSO allows a host system to behave as if it were reading or writing very large packets, and yet the actual packet size on the wire remains at the standard 1500 bytes.

On Linux systems, the status of TCO and TSO can be checked using the command `ethtool --show-offload` *interface*. TSO can be disabled with `ethtool --offload` *interface* `tso off`.

17.6 TCP simplex-talk

Here is a Java version of the simplex-talk server for TCP. As with the UDP version, we start by setting up the socket, here a `ServerSocket` called `ss`. This socket remains in the LISTEN state throughout. The main `while` loop then begins with the call `ss.accept()` at the start; this call blocks until an incoming connection is established, at which point it returns the connected child socket *s*. The `accept()` call models the TCP protocol behavior of waiting for three-way handshakes initiated by remote hosts and, for each, setting up a new connection.

Connections will be accepted from *all* IP addresses of the server host, *eg* the "normal" IP address, the loopback address 127.0.0.1 and, if the server is multihomed, any additional IP addresses. Unlike the UDP case (*16.1.3.2 UDP and IP addresses*), **RFC 1122** requires (§4.2.3.7) that server response packets always be sent from the same server IP address that the client first used to contact the server. (See *18 TCP Issues and Alternatives*, exercise 5.0 for an example of non-compliance.)

A server application can process these connected children either serially or in parallel. The stalk version here can handle both situations, either one connection at a time (`THREADING = false`), or by creating a new thread for each connection (`THREADING = true`). Either way, the connected child socket is turned over to `line_talker()`, either as a synchronous procedure call or as a new thread. Data is then read from the socket's associated `InputStream` using the ordinary `read()` call, versus the `receive()` used to read UDP packets. The main loop within `line_talker()` does not terminate until the client closes the connection (or there is an error).

In the serial, non-threading mode, if a second client connection is made while the first is still active, then data can be sent on the second connection but it sits in limbo until the first connection closes, at which point control returns to the `ss.accept()` call, the second connection is processed, and the second connection's data suddenly appears.

In the threading mode, the main loop spends almost all its time waiting in `ss.accept()`; when this returns a child connection we immediately spawn a new thread to handle it, allowing the parent process to go back to `ss.accept()`. This allows the program to accept multiple concurrent client connections, like the UDP version.

The code here serves as a very basic example of the creation of Java threads. The inner class `Talker` has a `run()` method, needed to implement the `runnable` interface. To start a new thread, we create a new `Talker` instance; the `start()` call then begins `Talker.run()`, which runs for as long as the client keeps the connection open. The file here is tcp_stalks.java.

```
/* THREADED simplex-talk TCP server */
/* can handle multiple CONCURRENT client connections */
/* newline is to be included at client side */

import java.net.*;
import java.io.*;

public class tstalks {

    static public int destport = 5431;
    static public int bufsize = 512;
    static public boolean THREADING = true;

    static public void main(String args[]) {
        ServerSocket ss;
        Socket s;
        try {
            ss = new ServerSocket(destport);
        } catch (IOException ioe) {
            System.err.println("can't create server socket");
            return;
        }
        System.err.println("server starting on port " + ss.getLocalPort());

        while(true) { // accept loop
            try {
                s = ss.accept();
            } catch (IOException ioe) {
                System.err.println("Can't accept");
```

(continues on next page)

(continued from previous page)

```
                break;
            }

            if (THREADING) {
                Talker talk = new Talker(s);
                (new Thread(talk)).start();
            } else {
                line_talker(s);
            }
        } // accept loop
    } // end of main

    public static void line_talker(Socket s) {
       int port = s.getPort();
        InputStream istr;
        try { istr = s.getInputStream(); }
        catch (IOException ioe) {
            System.err.println("cannot get input stream");        // most␣
↪likely cause: s was closed
            return;
        }
        System.err.println("New connection from <" +
            s.getInetAddress().getHostAddress() + "," + s.getPort() + ">");
        byte[] buf = new byte[bufsize];
        int len;

        while (true) {         // while not done reading the socket
            try {
                len = istr.read(buf, 0, bufsize);
            }
            catch (SocketTimeoutException ste) {
                System.out.println("socket timeout");
                continue;
            }
            catch (IOException ioe) {
                System.err.println("bad read");
                break;        // probably a socket ABORT; treat as a close
            }
            if (len == -1) break;              // other end closed gracefully
            String str = new String(buf, 0, len);
            System.out.print("" + port + ": " + str);  // str should contain␣
↪newline
        } //while reading from s


        try {istr.close();}
        catch (IOException ioe) {System.err.println("bad stream close");
↪return;}
        try {s.close();}
        catch (IOException ioe) {System.err.println("bad socket close");
↪return;}
        System.err.println("socket to port " + port + " closed");
```

(continues on next page)

(continued from previous page)

```
    } // line_talker

    static class Talker implements Runnable {
        private Socket _s;

        public Talker (Socket s) {
            _s = s;
        }

        public void run() {
            line_talker(_s);
        } // run
    }  // class Talker
}
```

17.6.1 The TCP Client

Here is the corresponding client tcp_stalkc.java. As with the UDP version, the default host to connect to is `localhost`. We first call `InetAddress.getByName()` to perform the DNS lookup. Part of the construction of the Socket object is the connection to the desired dest and destport. Within the main `while` loop, we use an ordinary `write()` call to write strings to the socket's associated `OutputStream`.

```
// TCP simplex-talk CLIENT in java

import java.net.*;
import java.io.*;

public class stalkc {

    static public BufferedReader bin;
    static public int destport = 5431;

    static public void main(String args[]) {
        String desthost = "localhost";
        if (args.length >= 1) desthost = args[0];
        bin = new BufferedReader(new InputStreamReader(System.in));

        InetAddress dest;
        System.err.print("Looking up address of " + desthost + "...");
        try {
            dest = InetAddress.getByName(desthost);
        }
        catch (UnknownHostException uhe) {
            System.err.println("unknown host: " + desthost);
            return;
        }
        System.err.println(" got it!");

        System.err.println("connecting to port " + destport);
        Socket s;
```

(continues on next page)

(continued from previous page)

```
        try {
            s = new Socket(dest, destport);
        }
        catch(IOException ioe) {
            System.err.println("cannot connect to <" + desthost + "," +␣
↪destport + ">");
            return;
        }

        OutputStream sout;
        try {
            sout = s.getOutputStream();
        }
        catch (IOException ioe) {
            System.err.println("I/O failure!");
            return;
        }

        //============================================================

        while (true) {
            String buf;
            try {
                buf = bin.readLine();
            }
            catch (IOException ioe) {
                System.err.println("readLine() failed");
                return;
            }
            if (buf == null) break;      // user typed EOF character

            buf = buf + "\n";           // protocol requires sender includes \n
            byte[] bbuf = buf.getBytes();

            try {
                sout.write(bbuf);
            }
            catch (IOException ioe) {
                System.err.println("write() failed");
                return;
            }
        } // while
    }
}
```

A Python3 version of the stalk client is available at tcp_stalkc.py.

Here are some things to try with THREADING=false in the server:

- start up two clients while the server is running. Type some message lines into both. Then exit the first client.
- start up the client before the server.

- start up the server, and then the client. Kill the server and then type some message lines to the client. What happens to the client? (It may take a couple message lines.)
- start the server, then the client. Kill the server and restart it. Now what happens to the client?

With THREADING=true, try connecting multiple clients simultaneously to the server. How does this behave differently from the first example above?

See also exercise 13.0.

17.7 TCP and `bind()`

The server version calls the `ServerSocket()` constructor, to create a socket in the LISTEN state; the local port must be specified here. The client version just calls `Socket()`, and the socket is then bound to a port by the operating system. It is also possible to create a client socket that is bound to a programmer-specified port; one application of this is to enable internal firewalls to identify the traffic class by source port. More commonly, client sockets are assigned *ephemeral* ports by the system. The Linux ephemeral port range can be found in `/proc/sys/net/ipv4/ip_local_port_range`; as of 2022 it is 32768 to 60999.

In C, sockets are created with `socket()`, bound to a port with `bind()`, and placed in the LISTEN state with `listen()` or else connected to a server with `connect()`. For client sockets, the call to `bind()` may be performed implicitly by `connect()`.

If two sockets on host A are connected via TCP to two *different* servers, B1 and B2, then it is possible that the operating system will assign the *same* local port to both sockets. On systems with an exceptional number of persistent outbound connections, such port reuse may be essential, as it is otherwise possible to run out of local ports. That said, port reuse is not an option if `bind()` is called explicitly, as at the time of the call to `bind()` the operating system does not yet know if the socket is to be used for LISTENing, for which a unique local port is necessary. To help deal with this, Linux has the socket option `IP_BIND_ADDRESS_NO_PORT`, which causes `bind()` to bind an IP address to the socket but defers port binding to a later `connect()`. Apple OS X provides the connectx() system call, which introduces similar functionality. See also this Cloudflare blog post.

Ephemeral ports were, originally, assigned more-or-less in sequence, skipping over values in use and ultimately wrapping around. This makes it easy, however, for adversaries to predict source-port numbers, so **RFC 6056** proposed making a first try at a new local-port number as follows:

> try0 = next_ephemeral + hash(local_addr, remote_addr, remote_port, secret_key)

If that is not available, subsequent tries incremented this value successively, with appropriate wrapping to stay within the designated ephemeral-port range. (See *18.3.1 ISNs and spoofing* for a related technique with ISN generation.)

This had the advantage that any *specific* remote socket would see the local ports incremented successively, which makes port reuse unlikely until the entire ephemeral-port range has been cycled through. However, a different remote socket would get another, unrelated, port-number sequence, making it very difficult for attackers to guess another connection's port.

Unfortunately, this elegant scheme introduced an unexpected problem: it enabled **fingerprinting** of the system that uses it, which lasted for the lifetime of the `secret_key`. This was done through the creation

of many "probe" connections, and observing the behavior of the local_port value; see [KK22] for details. The fix is to add a fast-changing timestamp to the hash arguments above, and to increment a failed port try by a random value between 1 and 7, rather than always by 1.

17.7.1 `netcat` again

As with UDP (*16.1.4 netcat*), we can use the `netcat` utility to act as either end of the TCP simplex-talk connection. As the client we can use

```
netcat localhost 5431
```

while as the server we can use

```
netcat -l -k 5431
```

Here (but not with UDP) the `-k` option causes the server to accept multiple connections in sequence. The connections are handled one at a time, as is the case in the stalk server above with `THREADING=false`.

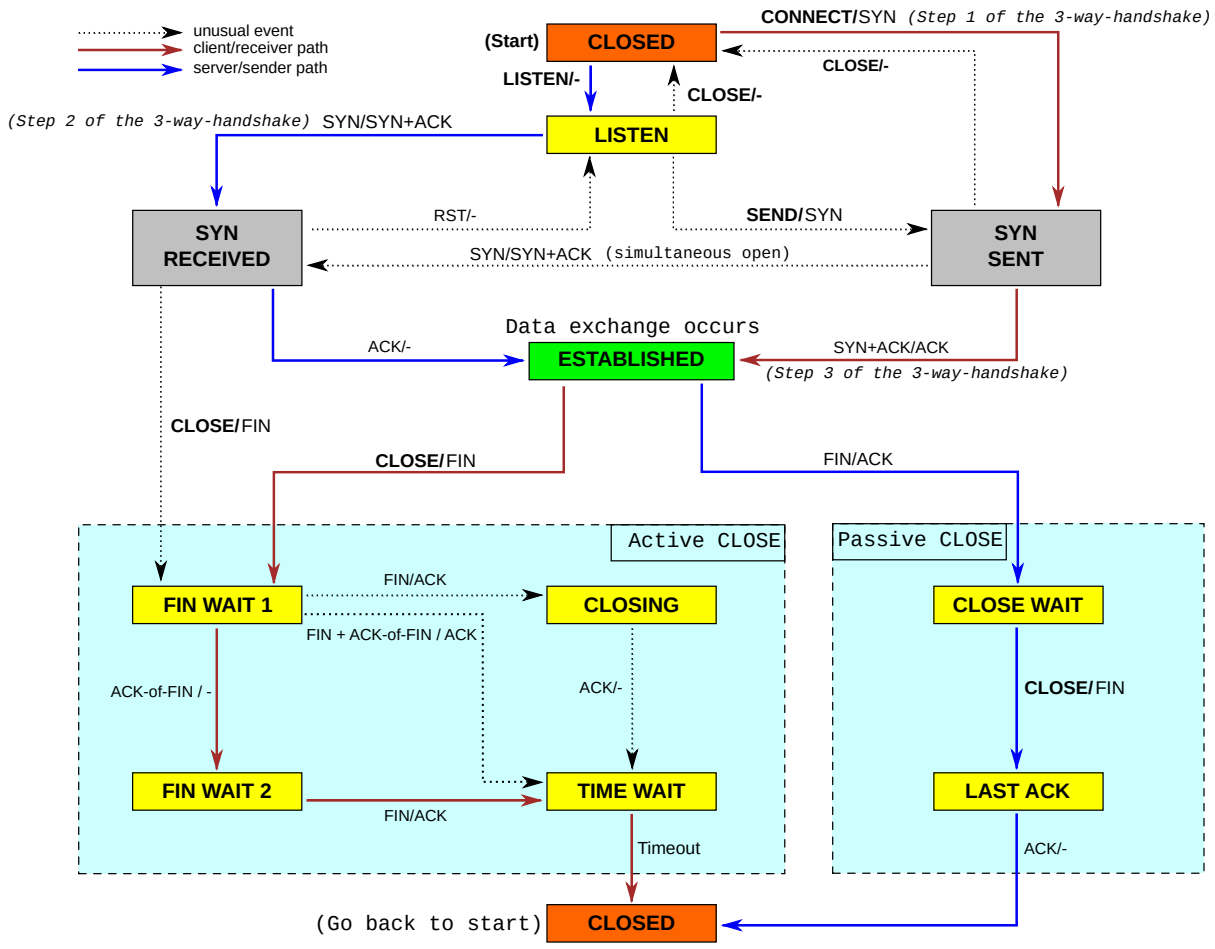
We can also use `netcat` to download web pages, using the HTTP protocol. The command below sends an HTTP GET request (version 1.1; **RFC 2616** and updates) to retrieve part of the website for this book; it has been broken over two lines for convenience.

```
echo -e 'GET /index.html HTTP/1.1\r\nHOST: intronetworks.cs.luc.edu\r\n'|
netcat intronetworks.cs.luc.edu 80
```

The `\r\n` represents the officially mandatory carriage-return/newline line-ending sequence, though `\n` will often work. The `index.html` identifies the file being requested; as `index.html` is the default it is often omitted, though the preceding `/` is still required. The webserver may support other websites as well via virtual hosting (*10.1.2 nslookup and dig*); the `HOST:` specification identifies to the server the specific site we are looking for. Version 2 of HTTP is described in **RFC 7540**; its primary format is binary. (For production command-line retrieval of web pages, cURL and wget are standard choices.)

17.8 TCP state diagram

A formal definition of TCP involves the **state diagram**, with conditions for transferring from one state to another, and responses to all packets from each state. The state diagram originally appeared in **RFC 793**; the following interpretation of the state diagram came from http://commons.wikimedia.org/wiki/File:Tcp_state_diagram_fixed.svg and was authored by Wikipedia users Sergiodc2, Marty Pauley, and DnetSvg. The blue arrows indicate the sequence of state transitions typically followed by the server; the brown arrows represent the client. Arrows are labeled with **event / action**; that is, we move from LISTEN to SYN_RECD upon receipt of a SYN packet; the action is to respond with SYN+ACK.

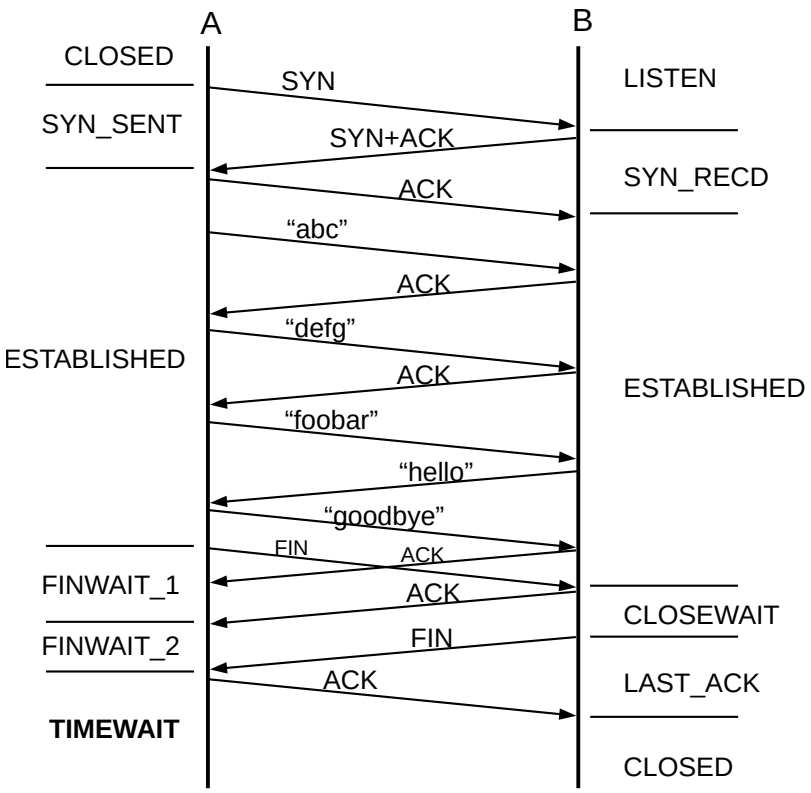


In general, this finite-state-machine approach to protocol specification has proven very effective, and is now used for most protocols. It makes it very clear to the implementer how the system should respond to each packet arrival. It is also a useful model for the implementation itself. Finally, we also observe that the TCP layer within an operating system cannot easily be modeled as anything *other* than a state machine; it must respond immediately to packet and program events, without indefinite waiting, as the operating system must go on to other things.

It is visually impractical to list every possible transition within the state diagram, full details are usually left to the accompanying text. For example, although this does not appear in the state diagram above, the per-state response rules of TCP require that in the ESTABLISHED state, if the receiver sends an ACK outside the current sliding window, then the correct response is to reply with one's own current ACK. This includes the case where the receiver *acknowledges data not yet sent*.

The ESTABLISHED state and the states below it are sometimes called the **synchronized** states, as in these states both sides have confirmed one another's ISN values.

Here is the ladder diagram for the 14-packet connection described above, this time labeled with TCP states.



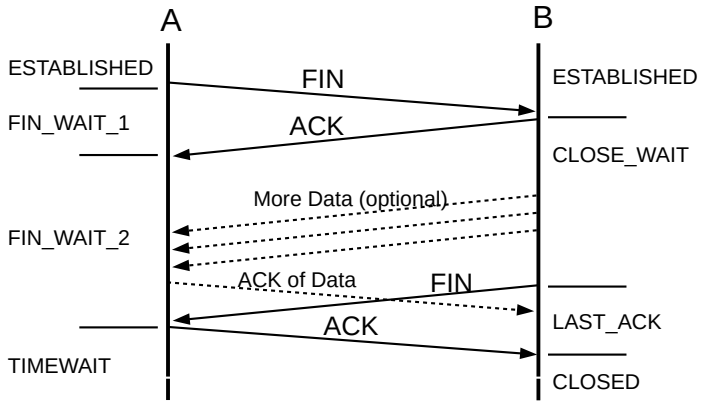
Although it essentially never occurs in practice, it is possible for each side to send the other a SYN, requesting a connection, **simultaneously** (that is, the SYNs cross on the wire). The telephony analogue occurs when each party dials the other simultaneously. On traditional land-lines, each party then gets a busy signal. On cell phones, your mileage may vary. With TCP, a single connection is created. With OSI TP4, two connections are created. The OSI approach is not possible in TCP, as a connection is determined only by the socketpair involved; if there is only one socketpair then there can be only one connection.

It is possible to view connection states under either Linux or Windows with `netstat -a`. Most states are ephemeral, exceptions being LISTEN, ESTABLISHED, TIMEWAIT, and CLOSE_WAIT. One sometimes sees large numbers of connections in CLOSE_WAIT, meaning that the remote endpoint has closed the connection and sent its FIN, but the process at your end has not executed `close()` on its socket. Often this represents a programming error; alternatively, the process at the local end is still working on something. Given a local port number p in state CLOSE_WAIT on a Linux system, the (privileged) command `lsof -i :`*p* will identify the process using port p.

The reader who is implementing TCP is encouraged to consult **RFC 793** and updates. For the rest of us, below are a few general observations about closing connections.

17.8.1 Closing a connection

The "normal" TCP close sequence is as follows:



Normal close

A's FIN is, in effect, a promise to B not to *send* any more. However, A must still be prepared to receive data from B, hence the optional data shown in the diagram. A good example of this occurs when A is sending a stream of data to B to be sorted; A sends FIN to indicate that it is done sending, and only then does B sort the data and begin sending it back to A. This can be generated with the command, on A, `cat thefile | ssh B sort`. That said, the presence of the optional B-to-A data above following A's FIN is relatively less common.

In the diagram above, A sends a FIN to B and receives an ACK, and then, later, B sends a FIN to A and receives an ACK. This essentially amounts to two separate two-way closure handshakes.

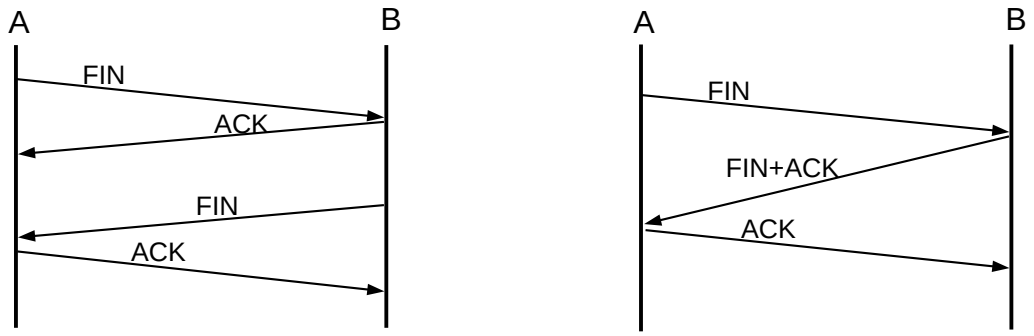
Either side may elect to close the connection, just as either party to a telephone call may elect to hang up. The first side to send a FIN – A in the diagram above – takes the **Active CLOSE** path; the other side takes the **Passive CLOSE** path. In the diagram, active-closer A moves from state ESTABLISHED to FIN_WAIT_1 to FIN_WAIT_2 (upon receipt of B's ACK of A's FIN), and then to TIMEWAIT and finally to CLOSED. Passive-closer B moves from ESTABLISHED to CLOSE_WAIT to LAST_ACK to CLOSED.

A **simultaneous close** – having both sides send each other FINs before receiving the other side's FIN – is a little more likely than a simultaneous open, earlier above, though still not very. Each side would send its FIN and move to state FIN_WAIT_1. Then, upon receiving each other's FIN packets, each side would send its final ACK and move to CLOSING. See exercises 5.0 and 6.0.

A TCP endpoint is **half-closed** if it has sent its FIN (thus promising not to send any more data) and is waiting for the other side's FIN; this corresponds to A in the diagram above in states FIN_WAIT_1 and FIN_WAIT_2. With the BSD socket library, an application can half-close its connection with the appropriate call to `shutdown()`.

Unrelatedly, A TCP endpoint is **half-open** if it is in the ESTABLISHED state, but during a lull in the exchange of packets the other side has rebooted; this has nothing to do with the close protocol. As soon as the ESTABLISHED side sends a packet, the rebooted side will respond with RST and the connection will be fully closed.

In the absence of the optional data from B to A after A sends its FIN, the closing sequence reduces to the left-hand diagram below:



Two TCP close scenarios with no B-to-A data

If B is ready to close immediately, it is possible for B's ACK and FIN to be combined, as in the right-hand diagram above, in which case the resultant diagram superficially resembles the connection-opening three-way handshake. In this case, A moves directly from FIN_WAIT_1 to TIMEWAIT, following the state-diagram link labeled "FIN + ACK-of-FIN". In theory this is rare, as the ACK of A's FIN is generated by the kernel but B's FIN cannot be sent until B's process is scheduled to run on the CPU. If the TCP layer adopts a policy of *immediately* sending ACKs upon receipt of any packet, this will never happen, as the ACK will be sent well before B's process can be scheduled to do anything. However, if B *delays* its ACKs slightly (and if it has no more data to send), then it is possible – and in fact not uncommon – for B's ACK and FIN to be sent together. Delayed ACKs, are, as we shall see below, a common strategy (*18.8 TCP Delayed ACKs*). To create the scenario of *17.4 TCP and WireShark*, it was necessary to introduce an artificial delay to prevent the simultaneous transmission of B's ACK and FIN.

17.8.2 Calling `close()`

Most network programming interfaces provide a `close()` method for ending a connection, based on the close operation for files. However, it usually closes bidirectionally and so models the TCP closure protocol rather imperfectly.

As we have seen in the previous section, the TCP close sequence is is followed more naturally if the active-closing endpoint calls `shutdown()` – promising not to send more, but allowing for continued receiving – before the final `close()`. Here is what *should* happen at the application layer if endpoint A of a TCP connection wishes to initiate the closing of its connection with endpoint B:

- A's application calls `shutdown()`, thereby promising not to send any more data. A's FIN is sent to B. A's application is expected to continue reading, however.
- The connection is now half-closed. On receipt of A's FIN, B's TCP layer knows this. If B's application attempts to read more data, it will receive an end-of-file indication (this is typically a `read()` or `recv()` operation that returns immediately with 0 bytes received).
- B's application is now done reading data, but it may or may not have more data to send. When B's application is done sending, it calls `close()`, at which point B's FIN is sent to A. Because the connection is already half-closed, B's `close()` is really a second half-close, ending further transmission by B.
- A's application keeps reading until it too receives an end-of-file indication, corresponding to B's FIN.

- The connection is now fully closed. No data has been lost.

It is sometimes the case that it is evident to A from the application protocol that B will not send more data. In such cases, A might simply call `close()` instead of `shutdown()`. This is risky, however, unless the protocol is crystal clear: if A calls `close()` and B later does send a little more data after all, or if B has already sent some data but A has not actually read it, A's TCP layer may send RST to B to indicate that not all B's data was received properly. **RFC 1122** puts it this way:

> If such a host issues a CLOSE call while received data is still pending in TCP, or if new data is received after CLOSE is called, its TCP SHOULD send a RST to show that data was lost.

If A's RST arrives at B before all of A's sent data has been processed by B's application, it is entirely possible that data sent by A will be lost, that is, will never be read by B.

In the BSD socket library, A can set the `SO_LINGER` option, which causes A's `close()` to block until A's data has been delivered to B (or until the `SO_LINGER` timeout, provided by the user when setting this option, has expired). However, `SO_LINGER` has no bearing on the issue above; post-close data from B to A will still cause A to send a RST.

In the simplex-talk program at *17.6 TCP simplex-talk*, the client does not call `shutdown()` (it implicitly calls `close()` when it exits). When the client is done, the server calls `s.close()`. However, the fact that there is no data at all sent from the server to the client prevents the problem discussed above.

It is sometimes the case that A is expected to send a large amount of data to B and then exit:

```
byte[] bbuf = byte[1000000];
...
sout.write(bbuf);               // Java OutputStream attached to the socket s
s.close()
```

In this case, the close() call is supposed to result in A sending all the data before actually terminating the connection. **RFC 793** puts it this way:

> Closing connections is intended to be a graceful operation in the sense that outstanding SENDs will be transmitted (and retransmitted), as flow control permits, until all have been serviced.

The Linux interpretation of this is given in the socket(7) man page:

> When the socket is closed as part of `exit(2)`, it always lingers in the background.

The linger *time* is not specified. If there is an explicit `close(2)` before `exit(2)`, the `SO_LINGER` status above determines TCP's behavior.

Alternatively, A can send the data and then attempt to read from the socket. A will receive an end-of-file indication (typically 0 bytes read) as soon as the other endpoint B closes. If B waits to close until it has read all the data, this end-of-file indication will mean it is safe for A to call `s.close()`. However, B might equally well call `shutdown()` immediately on startup, as it does not intend to write any data, in which case A's received end-of-file is **not** an indication it is safe to close.

See also exercises 13.0 and 15.0.

17.9 Epilog

At this point we have covered the basic mechanics of TCP. The next chapter discusses, among other things, some of the subtle issues TCP must deal with in order to maintain reliability.

17.10 Exercises

Exercises may be given fractional (floating point) numbers, to allow for interpolation of new exercises.

1.0. Experiment with the TCP version of simplex-talk. How does the server respond differently with threading enabled and without, if two simultaneous attempts to connect are made, from two different client instances?

2.0. Trace the states visited if nodes A and B attempt to create a TCP connection by *simultaneously* sending each other SYN packets, that then cross in the network. Draw the ladder diagram, and label the states on each side. Hint: there should be two pairs of crossing packets. A SYN+ACK counts, in the state diagram, as an ACK.

3.0. Suppose nodes A and B are each behind their own NAT firewall (*9.7 Network Address Translation*).

```
A ——— NAT_A ——— Internet ——— NAT_B ——— B
```

A and B attempt to connect to one another, using TCP; A uses source port 2000 and B uses 3000. A sends to the public IPv4 address of NAT_B, port 3000, and B sends to NAT_A, port 2000. Assume that neither NAT_A nor NAT_B changes the port numbers in outgoing packets, at least for the packets involved in this connection attempt.

(a). Suppose A sends a SYN packet to (NAT_B, 3000). It will be rejected at NAT_B, as the connection was not initiated by B. However, a short while later, B sends a SYN packet to (NAT_A, 2000). Explain why this second SYN packet *is* delivered to A.

(b). Now suppose A and B attempt to connect simultaneously, each sending a SYN to the other. Show that the connection succeeds.

4.0. When two nodes A and B simultaneously attempt to connect to one another using the OSI TP4 protocol, two bidirectional network connections are created (rather than one, as with TCP).

(a). Explain why this semantics is impossible with the existing TCP header. Hint: if a packet from ⟨A,port1⟩ arrives at ⟨B,port2⟩, how would the receiver tell to which of the two possible connections it belongs?

(b). Propose an additional field in the TCP header that would allow implementation of the TP4 semantics.

5.0. Simultaneous connection initiations are rare, but simultaneous connection termination is relatively common. How do two TCP nodes negotiate the simultaneous sending of FIN packets to one another? Draw the ladder diagram, and label the states on each side. Which node goes into TIMEWAIT state? Hint: there should be two pairs of crossing packets.

6.0. The state diagram at *17.8 TCP state diagram* shows a dashed path from FIN_WAIT_1 to TIMEWAIT on receipt of FIN+ACK. All FIN packets contain a valid ACK field, but that is not what is meant here. Under what circumstances is this direct arc from FIN_WAIT_1 to TIMEWAIT taken? Explain why this arc can never be used during simultaneous close. Hint: consider the ladder diagram of a "normal" close.

7.0. (a) Suppose you see multiple connections on your workstation in state FIN_WAIT_1. What is likely going on? Whose fault is it?

(b). What might be going on if you see connections languishing in state FIN_WAIT_2?

8.0. Suppose that, after downloading a file, the client host is unplugged from the network, so it can send no further packets. The server's connection is still in the ESTABLISHED state. In each case below, use the TCP state diagram to list all states that are reachable by the server.

(a). Before being unplugged, the client was in state ESTABLISHED; *ie* it had *not* sent the first FIN.

(b). Before being unplugged the client had sent its FIN, and moved to FIN_WAIT_1.

Eventually, the server connection here would in fact transition to CLOSED due to repeated timeouts. For this exercise, though, assume only transitions explicitly shown in the state diagram are allowed.

9.0. In *17.3 TCP Connection Establishment* we noted that RST packets had to have a valid SEQ value, but that "**RFC 793** does not require the RST packet's ACK value to match". There is an exception for RST packets arriving at state SYN-SENT: "the RST is acceptable if the ACK field acknowledges the SYN". Explain the reasoning behind this exception.

10.0. Suppose A and B create a TCP connection with ISN_A=20,000 and ISN_B=5,000. A sends three 1000-byte packets (Data1, Data2 and Data3 below), and B ACKs each. Then B sends a 1000-byte packet DataB to A and terminates the connection with a FIN. In the table below, fill in the SEQ and ACK fields for each packet shown.

A sends	B sends
SYN, ISN_A=20,000	
	SYN, ISN_B=5,000, ACK=_____
ACK, SEQ=_____, ACK=_____	
Data1, SEQ=_____, ACK=_____	
	ACK, SEQ=_____, ACK=_____
Data2, SEQ=_____, ACK=_____	
	ACK, SEQ=_____, ACK=_____
Data3, SEQ=_____, ACK=_____	
	ACK, SEQ=_____, ACK=_____
	DataB, SEQ=_____, ACK=_____
ACK, SEQ=_____, ACK=_____	
	FIN, SEQ=_____, ACK=_____

11.0. Suppose you are downloading a large file, and there is a progress bar showing how much of the file has been downloaded. For definiteness, assume the progress bar moves 1 mm for each megabyte received by the application, and the throughput averages 0.5 MB per second (so the progress bar normally advances at a rate of 0.5 mm/sec).

You see the progress bar stop advancing for an interval of time. Suddenly, it jumps forward 5 mm, and then resumes its steady 0.5 mm/sec advance.

(a). Explain the jump in terms of a lost packet and subsequent timeout and retransmission.

(b). Give an approximate value for the connection winsize, assuming only one packet was lost.

12.0. Suppose you are creating software for a streaming-video site. You want to limit the video read-ahead – the gap between how much has been downloaded and how much the viewer has actually watched – to approximately 1 MB; the server should pause in sending when necessary to enforce this. On the other hand, you do want the receiver to be able to read ahead by up to this much. You should assume that the TCP connection throughput will be higher than the actual video-data-consumption rate.

(a). Suppose the TCP window size happens to be exactly 1 MB. If the receiver simply reads each video frame from the TCP connection, displays it, and then pauses briefly before reading the next frame in accordance with the frame rate, explain how the flow-control mechanism of *18.10 TCP Flow Control* will achieve the desired effect.

(b). Applications, however, cannot control their TCP window size. Suppose the receiver application reads 1 MB ahead of what is being displayed, and then stops reading, until more can be displayed. Again, explain how the TCP flow-control mechanism will soon cause the sender to stop sending. (Specifically, the sender would never send more than winsize+1MB ahead of what was necessary.)

(It is also possible to implement sender pauses via an application-layer protocol.)

13.0. Modify the simplex-talk server of *17.6 TCP simplex-talk* so that `line_talker()` breaks out of the `while` loop as soon as it has printed the first string received (or simply remove the while loop). Once out of the `while` loop, the existing code calls `s.close()`.

(a). Start up the modified server, and connect to it with a client. Send a single message line, and use `netstat` to examine the TCP states of the client and server. What are these states?

(b). Send two message lines to the server. What are the TCP states of the client and server?

(c). Send three message lines to the server. Is there an error message at the client?

(d). Send two message lines to the server, while monitoring packets with WireShark. The WireShark filter expression `tcp.port == 5431` may be useful for eliminating irrelevant traffic. What FIN packets do you see? Do you see a RST packet?

14.0. Outline a scenario in which TCP endpoint A sends data to B and then calls `close()` on its socket, and after the connection terminates B has not received all the data, even though the network has not failed. In the style of *17.6.1 The TCP Client*, A's code might look like this:

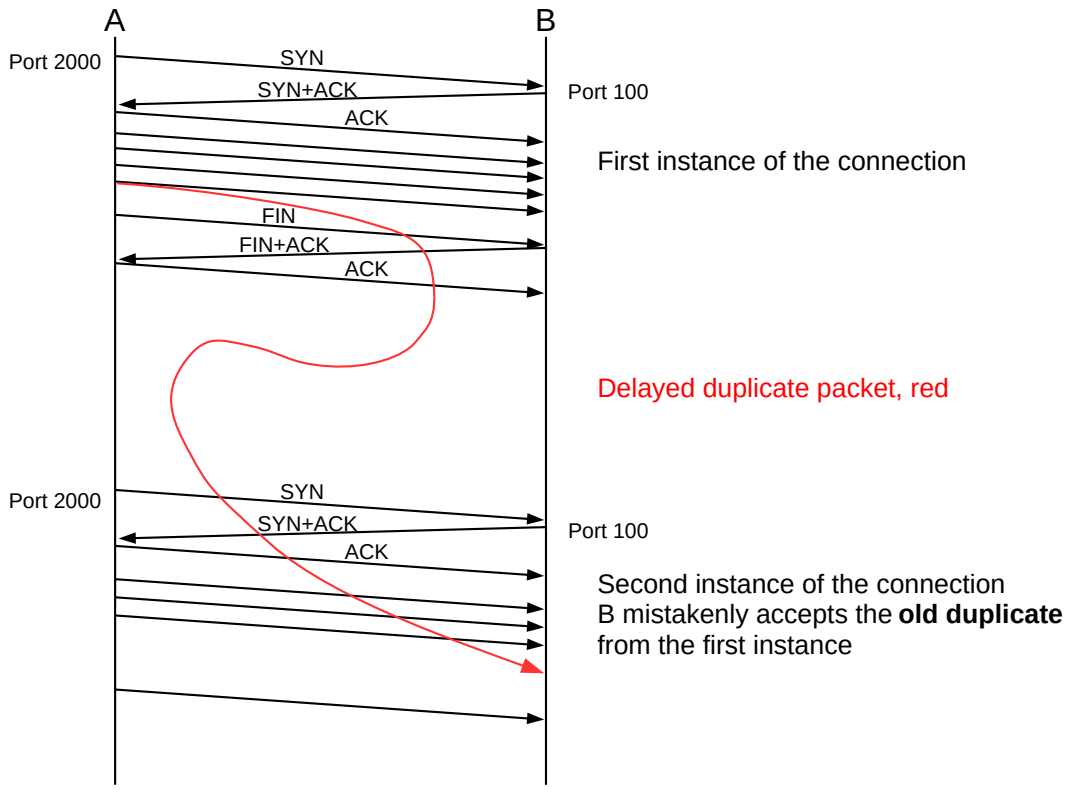
```
s = new Socket(dest, destport);
sout = s.getOutputStream();
sout.write(large_buffer)
s.close()
```

Hint: see *17.8.2 Calling close()*.

In this chapter we cover some issues relating to TCP reliability, some technical issues relating to TCP efficiency, and finally some outright alternatives to TCP.

18.1 TCP Old Duplicates

Conceptually, perhaps the most serious threat facing the integrity of TCP data is external old duplicates (*16.3 Fundamental Transport Issues*), that is, very late packets from a previous instance of the connection. Suppose a TCP connection is opened between A and B. One packet from A to B is duplicated and unduly delayed, with sequence number N. The connection is closed, and then another instance is reopened, that is, a connection is created using the same ports. At some point in the second connection, when an arriving packet with seq=N would be acceptable at B, the old duplicate shows up. Later, of course, B is likely to receive a seq=N packet from the new instance of the connection, but that packet will be seen by B as a duplicate (even though the data does not match), and (we will assume) be ignored.



For TCP, it is the actual sequence numbers, rather than the relative sequence numbers, that would have to match up. The diagram above ignores that.

As with TFTP, coming up with a possible scenario accounting for the generation of such a late packet is not easy. Nonetheless, many of the design details of TCP represent attempts to minimize this risk.

Solutions to the old-duplicates problem generally involve setting an upper bound on the lifetime of any packet, the MSL, as we shall see in the next section. T/TCP (*18.5 TCP Faster Opening*) introduced a connection-count field for this.

TCP is also vulnerable to sequence-number wraparound: arrival of an old duplicates from the *same* instance of the connection. However, if we take the MSL to be 60 seconds, sequence-number wrap requires sending 2^{32} bytes in 60 seconds, which requires a data-transfer rate in excess of 500 Mbps. TCP offers a fix for this (Protection Against Wrapped Segments, or PAWS), but it was introduced relatively late; we return to this in *18.4 Anomalous TCP scenarios*.

18.2 TIMEWAIT

The TIMEWAIT state is entered by whichever side initiates the connection close; in the event of a simultaneous close, both sides enter TIMEWAIT. It is to last for a time $2 \times$MSL, where MSL = Maximum Segment Lifetime is an agreed-upon value for the maximum lifetime on the Internet of an IP packet. Traditionally MSL was taken to be 60 seconds, but more modern implementations often assume 30 seconds (for a TIMEWAIT period of 60 seconds).

One function of TIMEWAIT is to solve the external-old-duplicates problem. TIMEWAIT requires that between closing and reopening a connection, a long enough interval must pass that any packets from the first instance will disappear. After the expiration of the TIMEWAIT interval, an old duplicate cannot arrive.

A second function of TIMEWAIT is to address the lost-final-ACK problem (*16.3 Fundamental Transport Issues*). If host A sends its final ACK to host B and this is lost, then B will eventually retransmit *its* final packet, which will be its FIN. As long as A remains in state TIMEWAIT, it can appropriately reply to a retransmitted FIN from B with a duplicate final ACK. As with TFTP, it is possible (though unlikely) for the final ACK to be lost as well as all the retransmitted final FINs sent during the TIMEWAIT period; should this happen, one side thinks the connection closed normally while the other side thinks it did not. See exercise 4.0.

TIMEWAIT only blocks reconnections for which both sides reuse the same port they used before. If A connects to B and closes the connection, A is free to connect again to B using a different port at A's end.

Conceptually, a host may have many old connections to the same port simultaneously in TIMEWAIT; the host must thus maintain for each of its ports a list of all the remote ⟨IP_address,port⟩ sockets currently in TIMEWAIT for that port. If a host is connecting as a client, this list likely will amount to a list of recently used ports; no port is likely to have been used twice within the TIMEWAIT interval. If a host is a server, however, accepting connections on a standardized port, and happens to be the side that initiates the active close and thus later goes into TIMEWAIT, then its TIMEWAIT list for that port can grow quite long.

Generally, busy servers prefer to be free from these bookkeeping requirements of TIMEWAIT, so many protocols are designed so that it is the client that initiates the active close. In the original HTTP protocol, version 1.0, the server sent back the data stream requested by the http GET message (*17.7.1 netcat again*), and indicated the end of this stream by closing the connection. In HTTP 1.1 this was fixed so that the client initiated the close; this required a new mechanism by which the server could indicate "I am done sending this file". HTTP 1.1 also used this new mechanism to allow the server to send back multiple files over one connection.

In an environment in which many short-lived connections are made from host A to the same port on server B, port exhaustion – having all ports tied up in TIMEWAIT – is a theoretical possibility. If A makes 1000

connections per second, then after 60 seconds it has gone through 60,000 available ports, and there are essentially none left. While this rate is high, early Berkeley-Unix TCP implementations often made only about 4,000 ports available to clients; with a 120-second TIMEWAIT interval, port exhaustion would occur with only 33 connections per second.

If you use ssh to connect to a server and then issue the `netstat -a` command on your own host (or, more conveniently, `netstat -a |grep -i tcp`), you should see your connection in ESTABLISHED state. If you close your connection and check again, your connection should be in TIMEWAIT.

18.3 The Three-Way Handshake Revisited

As stated earlier in *17.3 TCP Connection Establishment*, both sides choose an ISN; actual sequence numbers are the sum of the sender's ISN and the relative sequence number. There are two original reasons for this mechanism, and one later one (*18.3.1 ISNs and spoofing*). The original TCP specification, as clarified in **RFC 1122**, called for the ISN to be determined by a special **clock**, incremented by 1 every 4 microseconds.

The most basic reason for using ISNs is to detect duplicate SYNs. Suppose A initiates a connection to B by sending a SYN packet. B replies with SYN+ACK, but this is lost. A then times out and retransmits its SYN. B now receives A's second SYN while in state SYN_RECEIVED. Does this represent an entirely new request (perhaps A has suddenly restarted), or is it a duplicate? If A uses the clock-driven ISN strategy, B can tell (*almost* certainly) whether A's second SYN is new or a duplicate: only in the latter case will the ISN values in the two SYNs match.

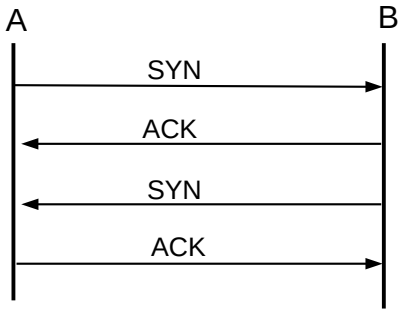
While there is no danger to data integrity if A sends a SYN, restarts, and sends the SYN again as part of a reopening the same connection, the arrival of a second SYN with a new ISN means that the original connection cannot proceed, because that ISN is now wrong. The receiver of the duplicate SYN should drop any connection state it has recorded so far, and restart processing the second SYN from scratch.

The clock-driven ISN also originally added a second layer of protection against external old duplicates. Suppose that A opens a connection to B, and chooses a clock-based ISN N_1. A then transfers M bytes of data, closed the connection, and reopens it with ISN N_2. If $N_1 + M < N_2$, then the old-duplicates problem *cannot occur*: all of the absolute sequence numbers used in the first instance of the connection are less than or equal to $N_1 + M$, and all of the absolute sequence numbers used in the second instance will be greater than N_2.

Early Berkeley-Unix implementations of the socket library often allowed a second connection meeting the above ISN requirement to be reopened *before* TIMEWAIT would have expired; this potentially addressed the problem of port exhaustion. We might call this **TIMEWAIT connection reuse**. Of course, if the first instance of the connection transferred data faster than the ISN clock rate, that is at more than 250,000 bytes/sec, then $N_1 + M$ would be greater than N_2, and TIMEWAIT would have to be enforced. But in the era in which TCP was first developed, sustained transfers exceeding 250,000 bytes/sec were not as common. Alternatively, the connection in TIMEWAIT might allow *incoming* connections that reuse the same ports, because in this case the host in TIMEWAIT can *choose* its own ISN to be greater than the final absolute sequence number of the previous instance of the connection. This second alternative is allowed by :rfc:1192:, §4.2.2.13.

The three-way handshake was extensively analyzed by Dalal and Sunshine in [DS78]. The authors noted that with a two-way handshake, the second side receives no confirmation that its ISN was correctly received.

The authors also observed that a four-way handshake – in which the ACK of ISN_A is sent separately from ISN_B, as in the diagram below – could fail if one side restarted.



Possible four-way handshake
Less secure than three-way version

For this failure to occur, assume that after sending the SYN in line 1, with ISN_{A1}, A restarts. The ACK in line 2 is either ignored or not received. B now sends its SYN in line 3, but A interprets this as a new connection request; it will respond after line 4 by sending a fifth, SYN packet containing a different ISN_{A2}. For B the connection is now ESTABLISHED, and if B acknowledges this fifth packet but fails to update its record of A's ISN, the connection will fail as A and B would have different notions of ISN_A.

18.3.1 ISNs and spoofing

The clock-based ISN proved to have a significant weakness: it often allowed an attacker to guess the ISN a remote host might use. It did not help any that an early version of Berkeley Unix, instead of incrementing the ISN 250,000 times a second, incremented it once a second, by 250,000 (plus something for each connection). By guessing the ISN a remote host would choose, an attacker might be able to mimic a local, trusted host, and thus gain privileged access.

Specifically, suppose host A trusts its neighbor B, and executes with privileged status commands sent by B; this situation was typical in the era of the `rhost` command. A authenticates these commands because the connection comes from B's IP address. The bad guy, M, wants to send packets to A so as to *pretend* to be B, and thus get a privileged command invoked. The connection only needs to be *started*; if the ruse is discovered after the command is executed, it is too late. M can easily send a SYN packet to A with B's IP address in the source-IP field; M can probably temporarily disable B too, so that A's SYN-ACK response, which is sent to B, goes unnoticed. What is harder is for M to figure out how to guess how to ACK ISN_A. But if A generates ISNs with a slowly incrementing clock, M can guess the pattern of the clock with previous connection attempts, and can thus guess ISN_A with a considerable degree of accuracy. So M sends SYN to A with B as source, A sends SYN-ACK to B containing ISN_A, and M *guesses* this value and sends ACK(ISN_A+1) to A, again with B listed in the IP header as source, followed by a single-packet command.

This TCP-layer IP-spoofing technique was first described by Robert T Morris in [RTM85]; Morris went on to launch the Internet Worm of 1988 using unrelated attacks. The IP-spoofing technique was used in the 1994 Christmas Day attack against UCSD, launched from Loyola's own apollo.it.luc.edu; the attack was associated with Kevin Mitnick though apparently not actually carried out by him. Mitnick was arrested a few months later.

RFC 1948, in May 1996, introduced a technique for introducing a degree of randomization in ISN selection, while still ensuring that the same ISN would not be used twice in a row for the same connection. The ISN is

to be the sum of the 4-µs clock, C(t), and a secure hash of the connection information as follows (compare with the local-port algorithm of *17.7 TCP and bind()*):

ISN = C(t) + hash(local_addr, local_port, remote_addr, remote_port, secret_key)

The `secret_key` value is a random value chosen by the host on startup. While M, above, can poll A for its current ISN, and can probably guess the hash function and the first four parameters above, without knowing the key it cannot determine (or easily guess) the ISN value A would have sent to B. Legitimate connections between A and B using the same port at each end, on the other hand, see the ISN increasing at the 4-µs rate, which potentially increases the chance of successful application of "TIMEWAIT connection reuse" as in *18.3 The Three-Way Handshake Revisited*.

RFC 5925 addresses spoofing and related attacks by introducing an optional TCP authentication mechanism: the TCP header includes an option containing a secure hash (*28.6 Secure Hashes*) of the rest of the TCP header and a shared secret key. The need for key management limits when this mechanism can be used; the classic use case is BGP connections between routers (*15 Border Gateway Protocol (BGP)*).

Another approach to the prevention of spoofing attacks is to ask sites and ISPs to refuse to forward outwards any IP packets with a source address not from within that site or ISP. If an attacker's ISP implements this, the attacker will be unable to launch spoofing attacks against the outside world. A concrete proposal can be found in **RFC 2827**. Unfortunately, it has been (as of 2015) almost entirely ignored.

See also the discussion of SYN flooding at *17.3 TCP Connection Establishment*, although that attack does not involve ISN manipulation.

18.4 Anomalous TCP scenarios

TCP, like any transport protocol, must address the transport issues in *16.3 Fundamental Transport Issues*.

As we saw above, TCP addresses the Duplicate Connection Request (Duplicate SYN) issue by noting whether the ISN has changed. This is handled at the kernel level by TCP, versus TFTP's application-level (and rather desultory) approach to handing Duplicate RRQs.

TCP addresses Loss of Final ACK through TIMEWAIT: as long as the TIMEWAIT period has not expired, if the final ACK is lost and the other side resends its final FIN, TCP will still be able to reissue that final ACK. TIMEWAIT in this sense serves a similar function to TFTP's DALLY state.

External Old Duplicates, arriving as part of a previous instance of the connection, are prevented by TIMEWAIT, and may also be prevented by the use of a clock-driven ISN.

Internal Old Duplicates, from the *same* instance of the connection, that is, sequence number wraparound, is only an issue for bandwidths exceeding 500 Mbps: only at bandwidths above that can 4 GB be sent in one 60-second MSL. TCP implementations now address this with PAWS: Protection Against Wrapped Segments (**RFC 1323**). PAWS adds a 32-bit "timestamp option" to the TCP header. The granularity of the timestamp clock is left unspecified; one tick must be small enough that sequence numbers cannot wrap in that interval (*eg* less than 3 seconds for 10,000 Mbps), and large enough that the timestamps cannot wrap in time MSL. On Linux systems the timestamp clock granularity is typically 1 to 10 ms; measurements on the author's systems have been 4 ms. With timestamps, an old duplicate due to sequence-number wraparound can now easily be detected.

The PAWS mechanism also requires ACK packets to echo back the sender's timestamp, in addition to including their own. This allows senders to accurately measure round-trip times.

Reboots are a potential problem as the host presumably has no record of what aborted connections need to remain in TIMEWAIT. TCP addresses this on paper by requiring hosts to implement Quiet Time on Startup: no new connections are to be accepted for 1*MSL. No known implementations actually do this; instead, they assume that the restarting process itself will take at least one MSL. This is no longer as certain as it once was, but serious consequences have not ensued.

18.5 TCP Faster Opening

If a client wants to connect to a server, send a request and receive an immediate reply, TCP mandates one full RTT for the three-way handshake before data can be delivered. This makes TCP one RTT slower than UDP-based request-reply protocols. There have been periodic calls to allow TCP clients to include data with the first SYN packet *and* have it be delivered immediately upon arrival – this is known as **accelerated open**.

If there will be a series of requests and replies, the simplest way to deliver the data without a handshake delay is to **pipeline** all the requests and replies over one persistent connection; the handshake delay then applies only to the first request. If the pipeline connection is idle for a long-enough interval, it may be closed, and then reopened later if necessary.

An early accelerated-open proposal was **T/TCP**, or TCP for Transactions, specified in **RFC 1644**. T/TCP introduced a **connection count** TCP option, called CC; each participant would include a 32-bit CC value in its SYN; each participant's own CC values were to be monotonically increasing. Accelerated open was allowed if the server side had the client's previous CC in a cache, and the new CC value was strictly greater than this cached value. This ensured that the new SYN was not a duplicate of an older SYN.

Unfortunately, this also bypasses the duplicate-SYN detection and modest validation of the client's IP address provided by the full three-way handshake, worsening the spoofing problem of *18.3.1 ISNs and spoofing*. If malicious host M wants to pretend to be B when sending a privileged request to A, all M has to do is send a single SYN+Data packet with an extremely large value for CC. Generally, the accelerated open succeeded as long as the CC value presented was larger that the value A had cached for B; it did not have to be larger by exactly 1.

18.5.1 TCP Fast Open

The TCP Fast Open (TFO) mechanism, described in **RFC 7413**, involves a secure "cookie" sent by the client as a TCP option; if a SYN+Data packet has a valid cookie, then the client has satisfactorily established its identity and the data may be released immediately to the receiving application.

Cookies can be either 4 or 16 bytes (probably, though not necessarily, corresponding to IPv4 and IPv6), and are requested by the client through a previous TCP handshake with a cookie-request option in the SYN packet. If a client includes a still-valid cookie in the SYN packet of a subsequent connection, the data accompanying that SYN packet is immediately released by the server to the application; the three-way handshake still completes but the data does not wait for it.

The same cookie can be reused multiple times. Cookies do have an expiration time, though, and also they are specific to the client IP address (though not to the TCP ports used). One implementation option is for

the server to use encryption (not hashing) of the client IP address; in this model the cookie expires when the encryption key expires.

Because cookies must be requested ahead of time, TCP Fast Open is not fundamentally faster than the connection-pipeline option above, except that holding a TCP connection open uses more resources than simply storing a cookie. One likely application for TCP Fast Open is in accessing web servers. Web clients and servers already keep a persistent connection open for a while, but often "a while" here amounts only to several seconds; TCP Fast Open cookies could remain active for much longer. Another potential use is for TCP-based DNS queries, for which there is no established mechanism for connection reuse.

A serious practical problem with TCP Fast Open is that some middleboxes (*9.7.2 Middleboxes*) remove TCP options they do not understand, or even block the connection attempt entirely. One consequence of this is that clients attempting to use TFO must log failures, and not attempt to reuse TFO again (at least for an appropriate time interval).

Also, SYN flooding attacks are still possible; for example, a large number of compromised clients can obtain cookies legitimately, and then each reuse their cookie many times in short order. Alternatively, the cookie from *one* client can be distributed to a large number of other hosts which then spoof the original client's IP address. To minimize the impact of such attacks, TFO requires that the fast-open option be ignored if the number of pending fast opens exceeds a given threshold. Connections would still open normally, but data would not be delivered to the server application until the three-way handshake completed.

Finally, TFO does introduce a small possibility of duplicate data delivery. Consider, for example, the following sequence:

1. Then client sends a SYN with valid TFO cookie, and some data
2. The ACK from the server is lost, or is never sent
3. The server processes the data
4. The server reboots
5. The client times out, and retransmits its SYN+cookie+data, which arrives at the server

The duplicate data arriving in the final step will be again processed by the server. This is not *likely*, but if either the client or the server cannot handle the possibility of duplication, then TFO should not be used. In particular, it must be possible to enable or disable TFO on a per-connection basis. Of course, if the request conveyed by the SYN+data is idempotent (*16.5.2 Sun RPC*), the duplication should not matter.

An alternative duplicate-data-delivery scenario involves the client sending a SYN+cookie+data packet and closing the connection. The client, then, goes into TIMEWAIT, but not the server. Meanwhile, the SYN+cookie+data packet somehow gets duplicated within the network, and this duplicate arrives at the server. This scenario (unlike the first) is not prevented by arranging for the server's cookie-generation key to become invalid after a reboot.

18.6 Path MTU Discovery

TCP connections are more efficient if they can keep large packets flowing between the endpoints. Once upon a time, TCP endpoints included just 512 bytes of data in each packet that was not destined for local delivery, to avoid fragmentation. TCP endpoints now typically engage in **Path MTU Discovery** which almost always

allows them to send larger packets; backbone ISPs are now usually able to carry 1500-byte packets. The **Path MTU** is the largest packet size that can be sent along a path without fragmentation.

The IPv4 strategy is to send an initial data packet with the IPv4 `DONT_FRAG` bit set. If the ICMP message `Frag_Required/DONT_FRAG_Set` comes back, or if the packet times out, the sender tries a smaller size. If the sender receives a TCP ACK for the packet, on the other hand, indicating that it made it through to the other end, it might try a larger size. Usually, the size range of 512-1500 bytes is covered by less than a dozen discrete values; the point is not to find the exact Path MTU but to determine a reasonable approximation rapidly.

IPv6 has no DONT_FRAG bit. Path MTU Discovery over IPv6 involves the periodic sending of larger packets; if the ICMPv6 message `Packet Too Big` is received, a smaller packet size must be used. **RFC 1981** has details.

18.7 TCP Sliding Windows

TCP implements sliding windows, in order to improve throughput. Window sizes are measured in terms of bytes rather than packets; this leaves TCP free to packetize the data in whatever segment size it elects. In the initial three-way handshake, each side specifies the maximum window size it is willing to accept, in the **Window Size** field of the TCP header. This 16-bit field can only go to 64 kB, and a 1 Gbps $\times$ 100 ms bandwidth$\times$delay product is 12 MB; as a result, there is a TCP **Window Scale** option that can also be negotiated in the opening handshake. The scale option specifies a power of 2 that is to be multiplied by the actual Window Size value. In the WireShark example above, the client specified a Window Size field of 5888 (= 4 $\times$ 1472) in the third packet, but with a Window Scale value of 2^6 = 64 in the first packet, for an effective window size of 64 $\times$ 5888 = 256 segments of 1472 bytes. The server side specified a window size of 5792 and a scaling factor of 2^5 = 32.

TCP may either transmit a bulk stream of data, using sliding windows fully, or it may send slowly generated interactive data; in the latter case, TCP may never have even one full segment outstanding.

In the following chapter we will see that a sender frequently reduces the actual TCP window size, in order to avoid congestion; the window size included in the TCP header is known as the **Advertised** Window Size. On startup, TCP does not send a full window all at once; it uses a mechanism called "slow start".

18.8 TCP Delayed ACKs

TCP receivers are allowed briefly to delay their ACK responses to new data. This offers perhaps the most benefit for interactive applications that exchange small packets, such as ssh and telnet. If A sends a data packet to B and expects an immediate response, delaying B's ACK allows the receiving *application* on B time to wake up and generate that application-level response, which can then be sent together with B's ACK. Without delayed ACKs, the kernel layer on B may send its ACK before the receiving application on B has even been scheduled to run. If response packets are small, that doubles the total traffic. The maximum ACK delay is 500 ms, according to **RFC 1122** and **RFC 2581**, though 200 ms is more common.

For bulk traffic, delayed ACKs simply mean that the ACK traffic volume is reduced. Because ACKs are cumulative, one ACK from the receiver can in principle acknowledge multiple data packets from the sender. Unfortunately, acknowledging too many data packets with one ACK can interfere with the self-clocking

aspect of sliding windows; the arrival of that ACK will then trigger a burst of additional data packets, which would otherwise have been transmitted at regular intervals. Because of this, the RFCs above specify that an ACK be sent, at a minimum, for every other data packet. For a discussion of how the sender should respond to delayed ACKs, see *19.2.1 TCP Reno Per-ACK Responses*.

The TCP ACK-delay time can usually be adjusted globally as a system parameter. Linux offers a `TCP_QUICKACK` option, as a flag to `setsockopt()`, to disable delayed ACKs on a per-connection basis, but only until the next TCP system call (including reads and writes). It must be invoked immediately after every receive operation to disable delayed ACKs entirely. This option is also not very portable.

The TSO option of *17.5 TCP Offloading*, used at the receiver, can also reduce the number of ACKs sent. If every two arriving data packets are consolidated via TSO into a single packet, then the receiver will appear to the sender to be acknowledging every other data packet. The ACK delay introduced by TSO is, however, usually quite small.

18.9 Nagle Algorithm

Like delayed ACKs, the Nagle algorithm (**RFC 896**) also attempts to improve the behavior of interactive small-packet applications. It specifies that a TCP endpoint generating small data segments – segments of less than the maximum size – should queue them until either it accumulates a full segment's worth or receives an ACK for all the previously sent packets (small or not). If the full-segment threshold is not reached at the sender, this means that only one segment will be sent per RTT, containing all the data generated during that RTT.

Bandwidth Conservation

Delayed ACKs and the Nagle algorithm both originated in a bygone era, when bandwidth was in much shorter supply than it is today. In **RFC 896**, John Nagle writes (in 1984, well before TCP Reno, *19 TCP Reno and Congestion Management*) "In general, we have not been able to afford the luxury of excess long-haul bandwidth that the ARPANET possesses, and our long-haul links are heavily loaded during peak periods. Transit times of several seconds are thus common in our network." Today, it is unlikely that a modest number of small packets would cause detectable, let alone significant, problems. That said, abandoning the Nagle algorithm has the potential to unleash onto the Internet backbone large numbers of small, mostly-header packets; [MM01] suggests "it would be a mistake to stop using it."

As an example, suppose A wishes to send to B packets containing consecutive letters, starting with "a". The application on A generates these every 100 ms, but the RTT is 501 ms. At T=0, A transmits "a". The application on A continues to generate "b", "c", "d", "e" and "f" at times 100 ms through 500 ms, but A does not send them immediately. At T=501 ms, ACK("a") arrives; at this point A transmits its backlogged "bcdef". The ACK for this arrives at T=1002, by which point A has queued "ghijk". The end result is that A sends a fifth as many separate packets as it would without the Nagle algorithm. If these letters are generated by a user typing them with telnet, and the ACKs also include the echoed responses, then if the user pauses the echoed responses will very soon catch up.

The Nagle algorithm does not always interact well with delayed ACKs. If an application generates a 2 KB transaction that is divided between a full-sized packet and a followup small packet, then the Nagle algorithm means that the second packet cannot be sent until the first is acknowledged. However, a receiver

using delayed ACKs may wait up to 500 ms to send the ACK that allows that second packet to be sent. This delays the entire transaction by the delayed-ACK time. Worse, this may happen to every one of a lengthy *series* of transactions. Internet Draft A Proposed Modification to Nagle's Algorithm addresses this by, in effect, always allowing senders to send *one* small packet without receiving an acknowledgment, to compensate for the possibility that a delayed-ACK receiver will need to receive that one small packet before it sends its ACK. More specifically, the modification is to forbid the sending of small packets as long as there is an earlier unacknowledged *small* packet outstanding, versus the original rule forbidding the sending of small packets as long as there are *any* earlier unacknowledged packets, small or not.

For other examples, see exercises 1.0 and 2.0; the first is an example of how the Nagle algorithm can have surprising user-interface consequences. The Nagle algorithm can usually be disabled on a per-connection basis, in the BSD socket library by calling `setsockopt()` with the `TCP_NODELAY` flag.

18.10 TCP Flow Control

It is possible for a TCP sender to send data faster than the receiver can process it. When this happens, a TCP receiver may reduce the advertised Window Size value of an open connection, thus informing the sender to switch to a smaller window size. This provides support for **flow control**.

The window-size reduction appears in the ACKs sent back by the receiver. A given ACK is not supposed to reduce the window size by so much that the upper end of the window gets smaller. A window might shrink from the byte range [20,000..28,000] to [22,000..28,000] but never to [20,000..26,000].

If a TCP receiver uses this technique to shrink the advertised window size to 0, this means that the sender may not send data. The receiver has thus informed the sender that, yes, the data was received, but that, no, more may not yet be sent. This corresponds to the ACK_{WAIT} suggested in *8.1.3 Flow Control*. Eventually, when the receiver is ready to receive data, it will send an ACK increasing the advertised window size again.

If the TCP sender has its window size reduced to 0, and the ACK from the receiver increasing the window is lost, then the connection would be deadlocked. TCP has a special feature specifically to avoid this: if the window size is reduced to zero, the sender sends dataless packets to the receiver, at regular intervals. Each of these "polling" packets elicits the receiver's current ACK; the end result is that the sender will receive the eventual window-enlargement announcement reliably. These "polling" packets are regulated by the so-called **persist** timer.

18.11 Silly Window Syndrome

The silly-window syndrome is a term for a scenario in which TCP transfers only small amounts of data at a time. Because TCP/IP packets have a minimum fixed header size of 40 bytes, sending small packets uses the network inefficiently. The silly-window syndrome can occur when either by the receiving application consuming data slowly or when the sending application generating data slowly.

As an example involving a slow-consuming receiver, suppose a TCP connection has a window size of 1000 bytes, but the receiving application consumes data only 10 bytes at a time, at intervals about equal to the RTT. The following can then happen:

- The sender sends bytes 1-1000. The receiving application consumes 10 bytes, numbered 1-10. The receiving TCP buffers the remaining 990 bytes and sends an ACK reducing the window size to 10,

per *18.10 TCP Flow Control*.

- Upon receipt of the ACK, the sender sends 10 bytes numbered 1001-1010, the most it is permitted. In the meantime, the receiving application has consumed bytes 11-20. The window size therefore remains at 10 in the next ACK.
- the sender sends bytes 1011-1020 while the application consumes bytes 21-30. The window size remains at 10.

The sender may end up sending 10 bytes at a time indefinitely. This is of no benefit to either side; the sender might as well send larger packets less often. The standard fix, set forth in **RFC 1122**, is for the receiver to use its ACKs to keep the window at 0 until it has consumed one full packet's worth (or half the window, for small window sizes). At that point the sender is invited – by an appropriate window-size advertisement in the next ACK – to send another full packet of data.

The silly-window syndrome can also occur if the sender is *generating* data slowly, say 10 bytes at a time. The Nagle algorithm, above, can be used to prevent this, though for interactive applications sending small amounts of data in separate but closely spaced packets may actually be useful.

18.12 TCP Timeout and Retransmission

When TCP sends a packet containing user data (this excludes ACK-only packets), it sets a timeout. If that timeout expires before the packet data is acknowledged, it is retransmitted. Acknowledgments are sent for every arriving data packet (unless Delayed ACKs are implemented, *18.8 TCP Delayed ACKs*); this amounts to receiver-side retransmit-on-duplicate of *8.1.1 Packet Loss*. Because ACKs are cumulative, and so a later ACK can replace an earlier one, lost ACKs are seldom a problem.

For TCP to work well for both intra-server-room and trans-global connections, with RTTs ranging from well under 1 ms to close to 1 second, the length of the timeout interval must *adapt*. TCP manages this by maintaining a running estimate of the RTT, EstRTT. In the original version, TCP then set TimeOut = 2×EstRTT (in the literature, the TCP TimeOut value is often known as RTO, for Retransmission TimeOut). EstRTT itself was a running average of periodically measured SampleRTT values, according to

$$\text{EstRTT} = \alpha \times \text{EstRTT} + (1-\alpha) \times \text{SampleRTT}$$

for a fixed α, $0<\alpha<1$. Typical values of α might be $\alpha=1/2$ or $\alpha=7/8$. For α close to 1 this is "conservative" in that EstRTT is slow to change. For α closer to 0, EstRTT is more volatile.

There is a potential RTT measurement ambiguity: if a packet is sent twice, the ACK received could be in response to the first transmission or the second. The Karn/Partridge algorithm resolves this: on packet loss (and retransmission), the sender

- Doubles Timeout
- Stops recording SampleRTT
- Uses the doubled Timeout as EstRTT when things resume

Setting TimeOut = 2×EstRTT proved too short during congestion periods and too long other times. Jacobson and Karels ([JK88]) introduced a way of calculating the TimeOut value based on the statistical variability of EstRTT. After each SampleRTT value was collected, the sender would also update EstDeviation according to

SampleDev = | SampleRTT – EstRTT |
EstDeviation = $\beta \times$ EstDeviation + (1-β)$\times$SampleDev

for a fixed β, $0<\beta<1$. Timeout was then set to EstRTT + 4$\times$EstDeviation. EstDeviation is an estimate of the so-called *mean deviation*; 4 mean deviations corresponds (for normally distributed data) to about 5 *standard* deviations. If the SampleRTT values were normally distributed (which they are not), this would mean that the chance that a non-lost packet would arrive outside the TimeOut period is vanishingly small.

For further details, see [JK88] and [AP99].

In most implementations, a TCP sender maintains just one retransmission timer no matter how many packets are outstanding. Here is the recommended timer-management algorithm, from **RFC 6298**:

- If a packet is sent and the timer is not running, restart it for time Timeout.
- If an ACK arrives that acknowledges all outstanding data, turn off the timer.
- If an ACK arrives that acknowledges new data, but not all outstanding data, reset the timer to time Timeout.

If the sender transmits a steady stream of packets, none of which is lost, the last clause will ensure that the timer never fires, which is as desired. However, if a series of earlier ACKs arrives slowly, but just fast enough to keep resetting the timer, a lost packet may not time out until some multiple of Timeout has elapsed; while not ideal, this is not considered serious. See exercise 6.0.

18.13 KeepAlive

There is no reason that a TCP connection should not be idle for a long period of time; ssh/telnet connections, for example, might go unused for days. However, there is the turned-off-at-night problem: a workstation might telnet into a server, and then be shut off (not shut down gracefully) at the end of the day. The connection would now be half-open, but the server would not generate any traffic and so might never detect this; the connection itself would continue to tie up resources.

KeepAlive in action

One evening long ago, when dialed up (yes, that long ago) into the Internet, my phone line disconnected while I was typing an email message in an ssh window. I dutifully reconnected, expecting to find my message in the file "dead.letter", which is what would have happened had I been disconnected while using the even-older tty dialup. Alas, nothing was there. I reconstructed my email as best I could and logged off.

The next morning, there was my lost email in a file "dead.letter", dated two hours after the initial crash! What had happened, apparently, was that the original ssh connection on the server side just hung there, half-open. Then, after two hours, KeepAlive kicked in, and aborted the connection. At that point ssh sent my mail program the HangUp signal, and the mail program wrote out what it had in "dead.letter".

To avoid this, TCP supports an optional **KeepAlive** mechanism: each side "polls" the other with a dataless packet. The original **RFC 1122** KeepAlive timeout was 2 hours, but this could be reduced to 15 minutes. If a connection failed the KeepAlive test, it would be closed.

Supposedly, some TCP implementations are not exactly **RFC 1122**-compliant: either KeepAlives are enabled by default, or the KeepAlive interval is much smaller than called for in the specification.

18.14 TCP timers

To summarize, TCP maintains the following four kinds of timers. All of them can be maintained by a single timer list, above.

- **TimeOut**: a per-segment timer; TimeOut values vary widely
- 2×MSL **TIMEWAIT**: a per-connection timer
- **Persist**: the timer used to poll the receiving end when winsize = 0
- **KeepAlive**, above

18.15 Variants and Alternatives

One alternative to TCP is UDP with programmer-implemented timout and retransmission; many RPC implementations (*16.5 Remote Procedure Call (RPC)*) do exactly this, with reasonable results. Within a LAN a static timeout of around half a second usually works quite well (unless the LAN has some tunneled links), and implementation of a simple timeout-retransmission mechanism is quite straightforward, although implementing adaptive timeouts as in *18.12 TCP Timeout and Retransmission* is a bit more complex. QUIC (*16.1.1 QUIC*) is an example of this strategy.

We here consider four other protocols. The first, MPTCP, is based on TCP itself. The second, SCTP, is a message-oriented alternative to TCP that is an entirely separate protocol. The last two, DCCP and QUIC, are attempts to create a TCP-like transport layer on top of UDP.

18.15.1 MPTCP

Multipath TCP, or MPTCP, allows connections to use multiple network interfaces on a host, either sequentially or simultaneously. MPTCP architectural principles are outlined in **RFC 6182**; implementation details are in **RFC 6824**.

To carry the actual traffic, MPTCP arranges for the creation of multiple standard-TCP **subflows** between the sending and receiving hosts; these subflows typically connect between different pairs of IP addresses on the respective hosts.

For example, a connection to a server can start using the client's wired Ethernet interface, and continue via Wi-Fi after the user has unplugged. If the client then moves out of Wi-Fi range, the connection might continue via a mobile network. Alternatively, MPTCP allows the parallel use of multiple Ethernet interfaces on both client and server for higher throughput.

MPTCP officially forbids the creation of multiple TCP connections between a single pair of interfaces in order to simulate Highspeed TCP (*22.5 Highspeed TCP*); **RFC 6356** spells out an MWTCP congestion-control algorithm to enforce this.

Suppose host A, with two interfaces with IP addresses A_1 and A_2, wishes to connect to host B with IP addresses B_1 and B_2. Connection establishment proceeds via the ordinary TCP three-way handshake, between one of A's IP addresses, say A_1, and one of B's, B_1. The SYN packets must each carry the `MP_CAPABLE` TCP option, to signal one another that MPTCP is supported. As part of the `MP_CAPABLE` option, A and B also exchange pseudorandom 64-bit connection keys, sent unencrypted; these will be used to sign later messages as in *28.6.1 Secure Hashes and Authentication*. This first connection is the initial subflow.

Once the MPTCP initial subflow has been established, additional subflow connections can be made. Usually these will be initiated from the client side, here A, though the B side can also do this. At this point, however, A does not know of B's address B_2, so the only possible second subflow will be from A_2 to B_1. New subflows will carry the `MP_JOIN` option with their initial SYN packets, along with digital signatures signed by the original connection keys verifying that the new subflow is indeed part of this MPTCP connection.

At this point A and B can send data to one another using both connections simultaneously. To keep track of data, each side maintains a 64-bit data sequence number, DSN, for the data it sends; each side also maintains a mapping between the DSN and the subflow sequence numbers. For example, A might send 1000-byte blocks of data alternating between the A_1 and A_2 connections; the blocks might have DSN values 10000, 11000, 12000, 13000, ... The A_1 subflow would then carry blocks 10000, 12000, *etc*, numbering these consecutively (perhaps 20000, 21000, ...) with its own sequence numbers. The sides exchange DSN mapping information with a `DSS` TCP option. This mechanism means that all data transmitted over the MWTCP connection can be delivered in the proper order, and that if one subflow fails, its data can be retransmitted on another subflow.

B can inform A of its second IP address, B_2, using the `ADD_ADDR` option. Of course, it is possible that B_2 is not directly reachable by A; for example, it might be behind a NAT router. But if B_2 *is* reachable, A can now open two more subflows A_1—B_2 and A_2—B_2.

All the above works equally well if either or both of A's addresses is behind a NAT router, simply because the NAT router is able to properly forward the subflow TCP connections. Addresses sent from one host to another, such as B's transmission of its address B_2, may be rendered invalid by NAT, but in this case A's attempt to open a connection to B_2 simply fails.

Generally, hosts can be configured to use multiple subflows in parallel, or to use one interface only as a backup, when the primary interface is unplugged or out of range. APIs have been proposed that allow an control over MPTCP behavior on a per-connection basis.

18.15.2 SCTP

The Stream Control Transmission Protocol, SCTP, is an entirely separate protocol from TCP, running directly above IP. It is, in effect, a message-oriented alternative to TCP: an application writes a sequence of messages and SCTP delivers each one as a unit, fragmenting and reassembling it as necessary. Like TCP, SCTP is connection-oriented and reliable. SCTP uses a form of sliding windows, and, like TCP, adjusts the window size to manage congestion.

An SCTP connection can support multiple **message streams**; the exact number is negotiated at startup. A retransmission delay in one stream never blocks delivery in other streams. Within each stream, SCTP messages are sequentially numbered, and are normally delivered in order of message number. A receiver can request, however, to receive messages immediately upon successful delivery, that is, potentially out of order. Either way, the data within each message is guaranteed to be delivered in order and without loss.

Internally, message data is divided into SCTP **chunks** for inclusion in packets. One SCTP packet can contain data chunks from different messages and different streams; packets can also contain control chunks.

Messages themselves can be quite large; there is no set limit. Very large messages may need to be received in multiple system calls (*eg* calls to `recvmsg()`).

SCTP supports an MPTCP-like feature by which each endpoint can use multiple network interfaces.

SCTP connections are set up using a four-way handshake, versus TCP's three-way handshake. The extra packet provides some protection against so-called SYN flooding (*17.3 TCP Connection Establishment*). The central idea is that if client A initiates a connection request with server B, then B allocates no resources to the connection until after B has received a response to its own message to A. This means that, at a minimum, A is a real host with a real IP address.

The full four-way handshake between client A and server B is, in brief, as follows:

- A sends B an INIT chunk (corresponding to SYN), along with a pseudorandom Tag_A.
- B sends A an `INIT ACK`, with Tag_B and a **state cookie**. The state cookie contains all the information B needs to allocate resources to the connection, and is digitally signed (*28.6.1 Secure Hashes and Authentication*) with a key known only to B. Crucially, B does **not** at this point allocate any resources to the incipient connection.
- A returns the state cookie to B in a `COOKIE ECHO` packet.
- B enters the ESTABLISHED state and sends a `COOKIE ACK` to A. Upon receipt, A enters the ESTABLISHED state.

When B receives the `COOKIE ECHO`, it verifies the signature. At this point B knows that it sent the cookie to A and received a response, so A must exist. Only then does B allocate memory resources to the connection. Spoofed INITs in the first step cost B essentially nothing.

The Tag_A and Tag_B in the first two packets are called **verification tags**. From this point on, B will include Tag_A in every packet it sends to A, and vice-versa. Although these tags are sent unencrypted, they nonetheless make it much harder for an attacker to inject data into the connection.

Data can be included in the third and fourth packets above; *ie* A can begin sending data after one RTT.

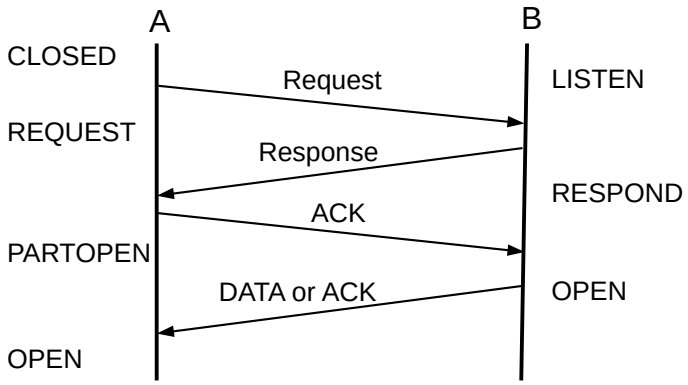
Unfortunately for potential SCTP applications, few if any NAT routers recognize SCTP; this limits the use of SCTP to Internet paths along which NAT is not used. In principle SCTP could simplify delivery of web pages, transmitting one page component per message, but lack of NAT support makes this infeasible. SCTP is also blocked by some middleboxes (*9.7.2 Middleboxes*) on the grounds that it is an unknown protocol, and therefore suspect. While this is not quite as common as the NAT problem, it is common enough to prevent by itself the widespread adoption of SCTP in the general Internet. SCTP *is* widely used for telecommunications signaling, both within and between providers, where NAT and recalcitrant middleboxes can be banished.

18.15.3 DCCP

As we saw in *16.1.2 DCCP*, DCCP is a UDP-based transport protocol that supports, among other things, connection establishment. While it is used much less often than TCP, it provides an alternative example of how transport can be done.

DCCP defines a set of distinct packet types, rather than TCP's independent packet flags; this disallows unforeseen combinations such as TCP SYN+RST. Connection establishment involves Request and Respond; data transmission involves Data, ACK and DataACK, and teardown involves CloseReq, Close and Reset. While one cannot have, for example, a Respond+ACK, Respond packets do carry an acknowledgment field.

Like TCP, DCCP uses a three-way handshake to open a connection; here is a diagram:



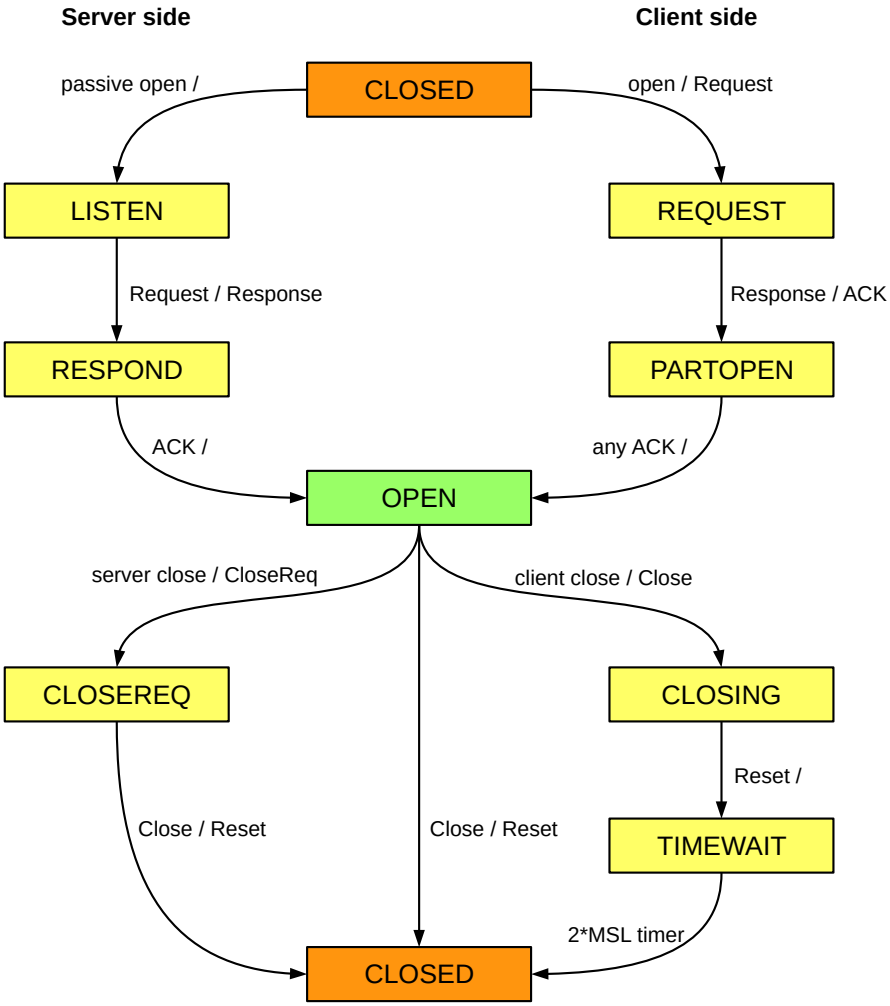
DCCP "three"-way handshake

The fourth packet, DATA or ACK, is not considered part of the handshake itself.

The OPEN state corresponds to TCP's ESTABLISHED state. Like TCP, each side chooses an ISN (not shown in the diagram). Because packet delivery is not reliable, and because ACKs are not cumulative, the client remains in PARTOPEN state until it has confirmed that the server has received its ACK of the server's Response. While in state PARTOPEN, the client can send ACK and DataACK but not ACK-less Data packets.

Packets are numbered sequentially. The numbering includes all packets, not just Data packets, and is by packet rather than by byte.

The DCCP state diagram is shown below. It is simpler than the TCP state diagram because DCCP does not support simultaneous opens.



DCCP State Diagram

To close a connection, one side sends Close and the other responds with Reset. Reset is used for normal close as well as for exceptional conditions. Because whoever sends the Close is then stuck with TIMEWAIT, the server side may send CloseReq to ask the client to send Close.

There are also two special packet formats, Sync and SyncAck, for resynchronizing sequence numbers after a burst of lost packets.

The other major TCP-like feature supported by DCCP is congestion control; see *21.3.3 DCCP Congestion Control*.

18.15.4 QUIC Revisited

Like DCCP, QUIC (see also *16.1.1 QUIC*) is a UDP-based transport protocol, aimed rather squarely at HTTP plus TLS (*29.5.2 TLS*). The fundamental goal of QUIC is to provide TLS encryption protection with as little overhead as possible, in a manner that competes fairly with TCP in the presence of congestion. Opening a QUIC connection, encryption included, takes a single RTT. QUIC can also be seen, however, as a

complete rewrite of TCP from the ground up; a reading of specific features sheds quite a bit of light on how the corresponding TCP features have fared over the past thirty-odd years. As of 2021, QUIC was finally edited to RFC status:

- **RFC 8999**: Version-independent Properties of QUIC
- **RFC 9000**: QUIC: A UDP-Based Multiplexed and Secure Transport (good overview)
- **RFC 9001**: Using TLS to Secure QUIC
- **RFC 9002**: QUIC Loss Detection and Congestion Control
- **RFC 9114**: HTTP/3 (which uses QUIC)

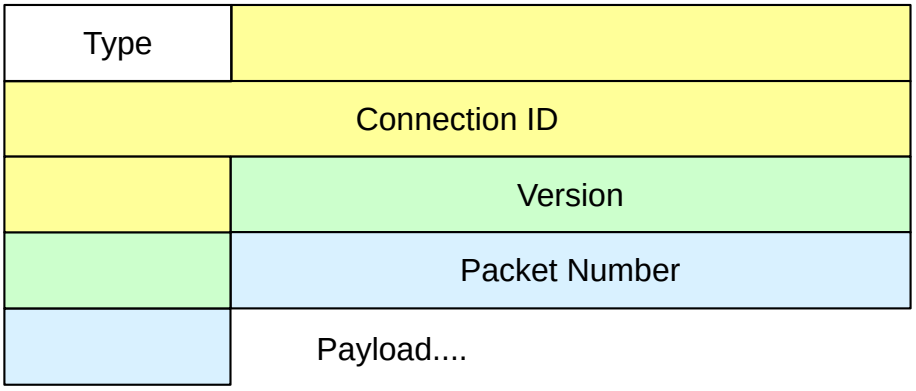
The move to base HTTP/3 on QUIC began officially in 2018; **RFC 9114** was published in June 2022. QUIC's standardization may represent the beginning of the end for TCP, given that most Internet connections are for HTTP or HTTPS. Still, many TCP design issues (*19 TCP Reno and Congestion Management*, *20 Dynamics of TCP*, *22 Newer TCP Implementations*) carry over very naturally to QUIC; a shift from TCP to QUIC should best be viewed as evolutionary. (And, by the same token, the 1995 standardization of IPv6 presumably represents the beginning of the end for IPv4, but that was over 25 years ago.)

The design of QUIC was influenced by the fate of SCTP above; the latter, as a new protocol above IP, was often blocked by overly security-conscious middleboxes (*9.7.2 Middleboxes*).

The fact that the QUIC layer resides within an application (or within a library) rather than within the kernel has meant that QUIC is able to evolve much faster than TCP. The long-term consequences of having the transport layer live outside the kernel are not yet completely clear, however; it may, for example, make it easier for users to install unfair congestion-management schemes.

18.15.4.1 Headers

We will start with the QUIC header. While there are some alternative forms, the basic header is diagrammed below, with a 1-byte `Type` field, an 8-byte `Connection ID`, and 4-byte `Version` and `Packet Number` fields.



Typical QUIC Long Header

Perhaps the most striking thing about this header is that 4-byte alignment – used consistently in the IPv4,

IPv6, UDP and TCP headers – has been completely abandoned. On most contemporary processors, the performance advantages of alignment are negligible; see the last paragraph at *9.1 The IPv4 Header*.

IP packets are identified as such by the Ethernet type field, and TCP and UDP packets are identified as such by the IPv4-header Protocol field. But QUIC packets are *not* identified as such by any flag in the preceding IP or UDP headers; there is in fact no place in those headers for a QUIC marker to go. QUIC appears to an observer as just another form of UDP traffic. This acts as a form of middlebox defense; QUIC packets cannot be identified as such in isolation. WireShark, sidebar below, identifies QUIC packets by looking at the whole history of the connection, and even then must make some (educated) guesses. Middleboxes could do that too, but it would take work.

The initial `Connection ID` consists of 64 random bits chosen by the client. The server, upon accepting the connection, may change the `Connection ID`; at that point the `Connection ID` is fixed for the lifetime of the connection. The `Connection ID` may be omitted for packets whose connection can be determined from the associated IP address and port values; this is signaled by the `Type` field. The `Connection ID` can also be used to migrate a connection to a different IP address and port, as might happen if a mobile device moves out of range of Wi-Fi and the mobile-data plan continues the communication. This may also happen if a connection passes through a NAT router. The NAT forwarding entry may time out (see the comment on UDP and inactivity at *9.7 Network Address Translation*), and the connection may be assigned a different outbound UDP port if it later resumes. QUIC uses the `Connection ID` to recognize that the reassigned connection is still the same one as before.

The `Version` field gets dropped as soon as the version is negotiated. As part of the version negotiation, a packet might have multiple version fields. Such packets put a *random* value into the low-order seven bits of the `Type` field, as a prevention against middleboxes' blocking unknown types. This way, aggressive middlebox behavior should be discovered early, before it becomes widespread.

QUIC-watching

QUIC packets can be observed in WireShark by using the filter string "quic". To generate QUIC traffic, use a Chromium-based browser and go to a Google-operated site, say, google.com. Often the only non-encrypted fields are the `Type` field and the packet number. It may be necessary to enable QUIC in the browser, done in Chrome via chrome://flags; see also chrome://net-internals.

The packet number can be reduced to one or two bytes once the connection is established; this is signaled by the `Type` field. Internally, QUIC uses packet numbers in the range 0 to 2^{62}; these internal numbers are not allowed to wrap around. The low-order 32 bits (or 16 bits or 8 bits) of the internal number are what is transmitted in the packet header. A packet receiver infers the high-order bits from the most recent acknowledgment.

The initial packet number is to be chosen randomly in the range 0 to 2^{32}–1025. This corresponds to TCP's use of Initial Sequence Numbers.

Use of 16-bit or 8-bit transmitted packet numbers is restricted to cases where there can be no ambiguity. At a minimum, this means that the number of outstanding packets (the QUIC winsize) cannot exceed 2^7–1 for 8-bit packet numbering or 2^{15}–1 for 16-bit packet numbering. These maximum winsizes represent the ideal case where there is no packet reordering; smaller values are likely to be used in practice. (See *8.5 Exercises*, exercise 9.0.)

18.15.4.2 Frames and streams

Data in a QUIC packet is partitioned into one or more **frames**. Each frame's data is prefixed by a simple frame header indicating its length and type. Some frames contain management information; frames containing higher-layer data are called STREAM frames. Each frame must be fully contained in one packet.

The application's data can be divided into multiple **streams**, depending on the application requirements. This is particularly useful with HTTP, as a client may request a large number of different resources (html, images, javascript, *etc*) simultaneously. Stream data is contained in STREAM frames. Streams are numbered, with Stream 0 reserved for the TLS cryptographic handshake. The HTTP/2 protocol has introduced its own notion of streams; these map neatly onto QUIC streams.

The two low-order bits of each stream number indicate whether the stream was initiated by the client or by the server, and whether it is bi- or uni-directional. This design decision means that either side can create a stream and send data on it immediately, *without negotiation*; this is important for reducing unnecessary RTTs.

Each individual stream is guaranteed in-order delivery, but there are no ordering guarantees between different streams. Within a packet, the data for a particular stream is contained in a frame for that stream.

One packet can contain stream frames for multiple streams. However, if a packet is lost, streams that have frames contained in that packet are blocked until retransmission. Other streams can continue without interruption. This creates an incentive for keeping separate streams in separate packets.

Stream frames contain the byte offset of the frame's block of stream data (starting from 0), to enable in-order stream reassembly. TCP, as we have seen, uses this byte-numbering approach exclusively, though starting with the Initial Sequence Number rather than zero. QUIC's stream-level numbering by byte is unrelated to its top-level numbering by packet.

In addition to stream frames, there are a large number of management frames. Here are a few of them:

- `RST_STREAM`: like TCP RST, but for one stream only.
- `MAX_DATA`: this corresponds to the TCP advertised window size. As with TCP, it can be reduced to zero to pause the flow of data and thereby implement flow control. There is also a similar `MAX_STREAM_DATA`, applying per stream.
- `PING` and `PONG`: to verify that the other endpoint is still responding. These serve as the equivalent of TCP KEEPALIVEs, among other things.
- `CONNECTION_CLOSE` and `APPLICATION_CLOSE`: these initiate termination of the connection; they differ only in that a `CONNECTION_CLOSE` might be accompanied by a QUIC-layer error or explanation message while an `APPLICATION_CLOSE` might be accompanied by, say, an HTTP error/explanation messge.
- `PAD`: to pad out the packet to a larger size.
- `ACK`: for acknowledgments, below.

18.15.4.3 Acknowledgments

QUIC assigns a new, sequential packet number (the `Packet ID`) to every packet, including retransmissions. TCP, by comparison, assigns sequence numbers to each byte. (QUIC stream frames do number data

by byte, as noted above.)

Lost QUIC packets are retransmitted, but with a new packet number. This makes it impossible for a receiver to send cumulative acknowledgments, as lost packets will never be acknowledged. The receiver handles this as below. At the sender side, the sender maintains a list of packets it has sent that are both unacknowledged and also not known to be lost. These represent the packets **in flight**. When a packet is retransmitted, its old packet number is removed from this list, as lost, and the new packet number replaces it.

To the extent possible given this retransmission-renumbering policy, QUIC follows the spirit of sliding windows. It maintains a state variable `bytes_in_flight`, corresponding to TCP's winsize, listing the total size of all the packets in flight. As with TCP, new acknowledgments allow new transmissions.

Acknowledgments themselves are sent in special acknowledgment frames. These begin with the number of the highest packet received. This is followed by a list of pairs, as long as will fit into the frame, consisting of the length of the next block of contiguous packets received followed by the length of the intervening gap of packets *not* received. The TCP Selective ACK (*19.6 Selective Acknowledgments (SACK)*) option is similar, but is limited to three blocks of received packets. It is quite possible that some of the gaps in a QUIC ACK frame refer to lost packets that were long since retransmitted with new packet numbers, but this does not matter.

The sender is allowed to skip packet numbers occasionally, to prevent the receiver from trying to increase throughput by acknowledging packets not yet received. Unlike with TCP, acknowledging an unsent packet is considered to be a fatal error, and the connection is terminated.

As with TCP, there is a delayed-ACK timer, but, while TCP's is typically 250 ms, QUIC's is 25 ms. QUIC also includes in each ACK frame the receiver's best estimate of the elapsed time between arrival of the most recent packet and the sending of the ACK it triggered; this allows the sender to better estimate the RTT. The primary advantage of the design decision not to reuse packet IDs is that there is never any ambiguity as to a retransmitted packet's RTT, as there is in TCP (*18.12 TCP Timeout and Retransmission*). Note, however, that because QUIC runs in a user process and not the kernel, it may not be able to respond immediately to an arriving packet, and so the time-delay estimate may be slightly short.

ACK frames are not themselves acknowledged. This means that, in a one-way data flow, the receiver may have no idea if its ACKs are getting through (a TCP receiver may be in the same situation). The QUIC receiver may send a PING frame to the sender, which will respond not only with a matching PONG frame but also an ACK frame acknowledging the receiver's recent acknowledgment packets.

QUIC adjusts its `bytes_in_flight` value to manage congestion, much as TCP manages its winsize (or more properly its `cwnd`, *19 TCP Reno and Congestion Management*) for the same purpose. Specifically, QUIC attempts to mimic the congestion response of TCP Cubic, *22.15 TCP CUBIC*, and so should in theory compete fairly with TCP Cubic connections. However, it is straightforward to arrange for QUIC to model the behavior of any other flavor of TCP (*22 Newer TCP Implementations*).

18.15.4.4 Connection handshake and TLS encryption

The opening of a QUIC connection makes use of the TLS handshake, *29.5.2 TLS*, specifically TLS v1.3, *29.5.2.4.3 TLS version 1.3*. A client wishing to connect sends a QUIC `Initial` packet, containing the TLS `ClientHello` message. The server responds (with a `ServerHello`) in a QUIC `Handshake` packet. (There is also a `Retry` packet, for special situations.) The TLS negotiation is contained in QUIC's

Stream 0. While the TLS and QUIC handshake rules are rather precise, there is as yet no formal state-diagram description of connection opening.

The `Initial` packet also contains a set of QUIC **transport parameters** declared unilaterally by the client; the server makes a similar declaration in its response. These parameters include, among other things, the maximum packet size, the connection's idle timeout, and initial value for `MAX_DATA`, above.

An important feature of TLS v1.3 is that, if the client has connected to the server previously and still has the key negotiated in that earlier session, it can use that old key to send an encrypted application-layer request (in a STREAM frame) immediately following the `Initial` packet. This is called **0-RTT** protection (or encryption). The advantage of this is that the client may receive an answer from the server within a single RTT, versus four RTTs for traditional TCP (one for the TCP three-way handshake, two for TLS negotiation, and one for the application request/reply). As discussed at *29.5.2.4.4 TLS v1.3 0-RTT mode*, requests submitted with 0-RTT protection must be idempotent, to prevent replay attacks.

Once the server's first `Handshake` packet makes it back to the client, the client is in possession of the key negotiated by the new session, and will encrypt everything using that going forward. This is known as the **1-RTT** key, and all further data is said to be 1-RTT protected. The negotiated key is initially calculated by the TLS layer, which then exports it to QUIC. The QUIC layer then encrypts the entire data portion of its packets, using the format of **RFC 5116**.

The QUIC header is not encrypted, but is still covered by an authentication checksum, making it impossible for middleboxes to rewrite anything. Such rewriting has been observed for TCP, and has sometimes complicated TCP evolution.

The type field of a QUIC packet contains a special code to mark 0-RTT data, ensuring that the receiver will know what level of protection is in effect.

When a QUIC server receives the `ClientHello` and sends off its `ServerHello`, it has not yet received any evidence that the client "owns" the IP address it claims to have; that is, that the client is not spoofing its IP address (*18.3.1 ISNs and spoofing*). Because of the idempotency restriction on responses to 0-RTT data, the server cannot give away privileges if spoofed in this way by a client. The server may, however, be an unwitting participant in a **traffic-amplification** attack, if the real client can trigger the sending by the server to a spoofed client of a larger response than the real client sends directly. The solution here is to require that the QUIC `Initial` packet, containing the `ClientHello`, be at least 1200 bytes. The server's `Handshake` response is likely to be smaller, and so represents no amplification of traffic.

To close the connection, one side sends a `CONNECTION_CLOSE` or `APPLICATION_CLOSE`. It may continue to send these in response to packets from the other side. When the other side receives the `CLOSE` packet, it should send its own, and then enter the so-called **draining** state. When the initiator of the close receives the other side's echoed `CLOSE`, it too will enter the draining state. Once in this state, an endpoint may not send any packets. The draining state corresponds to TCP's TIMEWAIT (*18.2 TIMEWAIT*), for the purpose of any lost final ACKs; it should last three RTT's. There is no need of a TIMEWAIT analog to prevent old duplicates, as a second QUIC connection will select a new `Connection ID`.

QUIC connection closing has no analog of TCP's feature in which one side sends FIN and the other continues to send data indefinitely, *17.8.1 Closing a connection*. This use of FIN, however, is allowed in bidirectional streams; the per-stream (and per-direction) FIN bit lives in the stream header. Alternatively, one side can send its request and close its stream, and the other side can then answer on a different stream.

18.16 Epilog

At this point we have covered the basic mechanics of TCP, but have one important topic remaining: how TCP manages its window size so as to limit congestion, while maintaining fairness. This turns out to be complex, and will be the focus of the next three chapters.

18.17 Exercises

Exercises may be given fractional (floating point) numbers, to allow for interpolation of new exercises.

1.0. A user moves the computer mouse and sees the mouse-cursor's position updated on the screen. Suppose the mouse-position updates are being transmitted over a TCP connection with a relatively long RTT. The user attempts to move the cursor to a specific point. How will the user perceive the mouse's motion

(a). with the Nagle algorithm
(b). without the Nagle algorithm

2.0. Host A sends two single-byte packets, one containing "x" and the other containing "y", to host B. A implements the Nagle algorithm and B implements delayed ACKs, with a 500 ms maximum delay. The RTT is negligible. How long does the transmission take? Draw a ladder diagram.

3.0. Suppose you have fallen in with a group that wants to add to TCP a feature so that, if A and B1 are connected, then B1 can **hand off** its connection to a different host B2; the end result is that A and B2 are connected and A has received an uninterrupted stream of data. Either A or B1 can initiate the handoff.

(a). Suppose B1 is the host to send the final FIN (or HANDOFF) packet to A. How would you handle appropriate analogues of the TIMEWAIT state for host B1? Does the fact that A is continuing the connection, just not with B1, matter?
(b). Now suppose A is the party to send the final FIN/HANDOFF, to B1. What changes to TIMEWAIT would have to be made at A's end? Note that A may potentially hand off the connection again and again, *eg* to B3, B4 and then B5.

4.0. Suppose A connects to B via TCP, and sends the message "Attack at noon", followed by FIN. Upon receiving this, B is sure it has received the entire message.

(a). What can A be sure of upon receiving B's own FIN+ACK?
(b). What can B be sure of upon receiving A's final ACK?
(c). What is A not absolutely sure of after sending its final ACK?

5.0. Host A connects to the Internet via Wi-Fi, receiving IPv4 address 10.0.0.2, and then opens a TCP

connection *conn1* to remote host B. After *conn1* is established, A's Ethernet cable is plugged in. A's Ethernet interface receives IP address 10.0.0.3, and A automatically selects this new Ethernet connection as its default route. **Assume** that A now starts using 10.0.0.3 as the source address of packets it sends as part of *conn1* (contrary to **RFC 1122**).

Assume also that A's TCP implementation is such that when a packet arrives from $\langle B_{IP}, B_{port} \rangle$ to $\langle A_{IP}, A_{port} \rangle$ and this socketpair is to be matched to an existing TCP connection, the field A_{IP} is allowed to be any of A's IP addresses (that is, either 10.0.0.2 or 10.0.0.3); it does not have to match the IP address with which the connection was originally negotiated.

(a). Explain why *conn1* will now fail, as soon as any packet is sent from A. Hint: the packet will be sent from 10.0.0.3. What will B send in response? In light of the second assumption, how will A react to B's response packet?

(The author regularly sees connections fail this way. Perhaps some justification for this behavior is that, at the time of establishment of *conn1*, A was not yet multihomed.)

(b). Now suppose all four fields of the socketpair ($\langle B_{IP}, B_{port} \rangle$, $\langle A_{IP}, A_{port} \rangle$) are used to match an incoming packet to its corresponding TCP connection. The connection *conn1* still fails, though not as immediately. Explain what happens.

See also *10.2.5 ARP and multihomed hosts*, *9 IP version 4* exercise 4.0, and *13 Routing-Update Algorithms* exercise 16.0.

6.0. Draw a ladder diagram showing a lost packet transmitted at time T, and yet the retransmission timer does not go off until at least T + 3*Timeout (there is nothing special about 3 here). Assume that the algorithm of *18.12 TCP Timeout and Retransmission* is used. Hint: show a series of ACKs for *previous* packets arriving at intervals of just under Timeout, causing a series of resets of the timer.

19 TCP RENO AND CONGESTION MANAGEMENT

This chapter addresses how TCP manages congestion, both for the connection's own benefit (to improve its throughput) and for the benefit of other connections as well (which may result in our connection *reducing* its own throughput). Early work on congestion culminated in 1990 with the flavor of TCP known as **TCP Reno**. The congestion-management mechanisms of TCP Reno remain the dominant approach on the Internet today, though alternative TCPs are an active area of research and we will consider a few of them in *22 Newer TCP Implementations*.

The central TCP mechanism here is for a connection to adjust its window size. A smaller winsize means fewer packets are out in the Internet at any one time, and less traffic means less congestion. A larger winsize means better throughput, up to a point. All TCPs reduce winsize when congestion is apparent, and increase it when it is not. The trick is in figuring out when and by how much to make these winsize changes. Many of the improvements to TCP have come from mining more and more information from the stream of returning ACKs.

The Anternet

The Harvester Ant *Pogonomyrmex barbatus* uses a mechanism related to TCP Reno to "decide" how many ants should be out foraging at any one time [PDG12]. The rate of ants leaving the nest to forage is closely tied to the rate of returning foragers; if foragers return quickly (meaning more food is available), the total number of foragers will increase (like the increasing winsize below). The ant algorithm is probabilistic, however, while most TCP algorithms are deterministic.

Recall Chiu and Jain's definition from *1.7 Congestion* that the "knee" of congestion occurs when the queue first starts to grow, and the "cliff" of congestion occurs when packets start being dropped. Congestion can be managed at either point, though dropped packets can be a significant waste of resources. Some newer TCP strategies attempt to take action at the congestion knee (starting with *22.6 TCP Vegas*), but TCP Reno is a cliff-based strategy: packets must be lost before the sender reduces the window size.

In *25 Quality of Service* we will consider some router-centric alternatives to TCP for Internet congestion management. However, for the most part these have not been widely adopted, and TCP is all that stands in the way of Internet congestive collapse.

The first question one might ask about TCP congestion management is just how did it get this job? A TCP sender is expected to monitor its transmission rate so as to *cooperate* with other senders to reduce overall congestion among the routers. While part of the goal of every TCP node is good, stable performance for its own connections, this emphasis on end-user cooperation introduces the prospect of "cheating": a host might be tempted to maximize the throughput of its own connections at the expense of others. Putting TCP nodes in charge of congestion among the core routers is to some degree like putting the foxes in charge of the henhouse. More accurately, such an arrangement has the potential to lead to the Tragedy of the Commons. Multiple TCP senders share a common resource – the Internet backbone – and while the backbone is most efficient if every sender cooperates, each individual sender can improve its own situation by sending faster than allowed. Indeed, one of the arguments used by virtual-circuit routing adherents is that it provides support for the implementation of a wide range of congestion-management options under control of a central authority.

Nonetheless, TCP has been quite successful at distributed congestion management. In part this has been because system vendors do have an incentive to take the big-picture view, and in the past it has been quite difficult for individual users to replace their TCP stacks with rogue versions. Another factor contributing to TCP's success here is that most bad TCP behavior requires cooperation at the *server* end, and most server managers have an incentive to behave cooperatively. Servers generally want to distribute bandwidth fairly among their multiple clients, and – theoretically at least – a server's ISP could penalize misbehavior. So far, at least, the TCP approach has worked remarkably well.

19.1 Basics of TCP Congestion Management

TCP's congestion management is **window-based**; that is, TCP adjusts its window size to adapt to congestion. The window size can be thought of as the number of packets out there in the network; more precisely, it represents the number of packets and ACKs either in transit or enqueued. An alternative approach often used for real-time systems is **rate-based** congestion management, which runs into an unfortunate difficulty if the sending rate momentarily happens to exceed the available rate.

In the very earliest days of TCP, the window size for a TCP connection came from the AdvertisedWindow value suggested by the receiver, essentially representing how many packet buffers it could allocate. This value is often quite large, to accommodate large bandwidth×delay products, and so is often reduced out of concern for congestion. When winsize is adjusted downwards for this reason, it is generally referred to as the **Congestion Window**, or `cwnd` (a variable name first appearing in Berkeley Unix). Strictly speaking, winsize = min(`cwnd`, AdvertisedWindow). In newer TCP implementations, the variable `cwnd` may actually be used to mean the sender's estimate of the number of packets in flight; see the sidebar at *19.4 TCP Reno and Fast Recovery*.

If TCP is sending over an idle network, the per-packet RTT will be $\text{RTT}_{\text{noLoad}}$, the travel time with no queuing delays. As we saw in *8.3.2 RTT Calculations*, (RTT–$\text{RTT}_{\text{noLoad}}$) is the time each packet spends in the queue. The path bandwidth is winsize/RTT, and so the number of packets in queues is winsize × (RTT–$\text{RTT}_{\text{noLoad}}$) / RTT. Usually all the queued packets are at the router at the head of the bottleneck link. Note that the sender can calculate this number (assuming we can estimate $\text{RTT}_{\text{noLoad}}$; the most common approach is to assume that the smallest RTT measured corresponds to $\text{RTT}_{\text{noLoad}}$).

TCP's self-clocking (*ie* that new transmissions are paced by returning ACKs) guarantees that, again assuming an otherwise idle network, the queue will build only at the bottleneck router. Self-clocking means that the rate of packet transmissions is equal to the available bandwidth of the bottleneck link. There are some spikes when a burst of packets is sent (*eg* when the sender increases its window size), but in the steady state self-clocking means that packets accumulate only at the bottleneck.

We will return to the case of the *non*-otherwise-idle network in the next chapter, in *20.2 Bottleneck Links with Competition*.

The "optimum" window size for a TCP connection would be bandwidth × $\text{RTT}_{\text{noLoad}}$. With this window size, the sender has exactly filled the transit capacity along the path to its destination, and has used none of the queue capacity.

Actually, TCP Reno does not do this.

Instead, TCP Reno does the following:

- guesses at a reasonable initial window size, using a form of polling

- slowly increases the window size if no losses occur, on the theory that maximum available throughput may not yet have been reached
- rapidly decreases the window size otherwise, on the theory that if losses occur then drastic action is needed

In practice, this usually leaves TCP's window size well above the theoretical "optimum".

One interpretation of TCP's approach is that there is a time-varying "ceiling" on the number of packets the network can accept. Each sender tries to stay near but just below this level. Occasionally a sender will overshoot and a packet will be dropped somewhere, but this just teaches the sender a little more about where the network ceiling is. More formally, this ceiling represents the largest `cwnd` that does not lead to packet loss, *ie* the `cwnd` that at that particular moment completely fills but does not overflow the bottleneck queue. We have reached the ceiling when the queue is full.

In Chiu and Jain's terminology, the far side of the ceiling is the "cliff", at which point packets are lost. TCP tries to stay above the "knee", which is the point when the queue first begins to be persistently utilized, thus keeping the queue at least partially occupied; whenever it sends too much and falls off the "cliff", it retreats.

The ceiling concept is often useful, but not necessarily as precise as it might sound. If we have reached the ceiling by *gradually* expanding the sliding-windows window size, then winsize will be as large as possible. But if the sender suddenly releases a burst of packets, the queue may fill and we will have reached a "temporary ceiling" without fully utilizing the transit capacity. Another source of ceiling ambiguity is that the bottleneck link may be shared with other connections, in which case the ceiling represents our connection's particular share, which may fluctuate greatly with time. Finally, at the point when the ceiling is reached, the queue is *full* and so there are a considerable number of packets waiting in the queue; it is not possible for a sender to pull back instantaneously.

It is time to acknowledge the existence of different versions of TCP, each incorporating different congestion-management algorithms. The two we will start with are **TCP Tahoe** (1988) and **TCP Reno** (1990); the names Tahoe and Reno were originally the codenames of the Berkeley Unix distributions that included these respective TCP implementations. The ideas behind TCP Tahoe came from a 1988 paper by Jacobson and Karels [JK88]; TCP Reno then refined this a couple years later. TCP Reno is still in widespread use over twenty years later, and is still the undisputed TCP reference implementation, although some modest improvements (NewReno, SACK) have crept in.

A common theme to the development of improved implementations of TCP is for one end of the connection (usually the sender) to extract greater and greater amounts of information from the packet flow. For example, TCP Tahoe introduced the idea that duplicate ACKs likely mean a lost packet; TCP Reno introduced the idea that returning duplicate ACKs are associated with packets that have successfully been transmitted but follow a loss. TCP Vegas (*22.6 TCP Vegas*) introduced the fine-grained measurement of RTT, to detect when $\text{RTT} > \text{RTT}_{\text{noLoad}}$.

It is often helpful to think of a TCP sender as having breaks between successive windowfuls; that is, the sender sends `cwnd` packets, is briefly idle, and then sends another `cwnd` packets. The successive windowfuls of packets are often called **flights**. The existence of any separation between flights is, however, not guaranteed.

19.1.1 The Somewhat-Steady State

We will begin with the state in which TCP has established a reasonable guess for `cwnd`, comfortably below the Advertised Window Size, and which largely appears to be working. TCP then engages in some fine-tuning. This TCP "steady state" – steady here in the sense of regular oscillation – is usually referred to as the **congestion avoidance** phase, though all phases of the process are ultimately directed towards avoidance of congestion. The central strategy is that when a packet is lost, `cwnd` should decrease rapidly, but otherwise should increase "slowly". This leads to slow oscillation of `cwnd`, which over time allows the average `cwnd` to adapt to long-term changes in the network capacity.

As TCP finishes each windowful of packets, it notes whether a loss occurred. The `cwnd`-adjustment rule introduced by TCP Tahoe and [JK88] is the following:

- if there were no losses in the previous windowful, `cwnd` = `cwnd` +1
- if packets were lost, `cwnd` = `cwnd`/2

We are informally measuring `cwnd` in units of full packets; strictly speaking, `cwnd` is measured in bytes and is incremented by the maximum TCP segment size.

This strategy here is known as **Additive Increase, Multiplicative Decrease**, or AIMD; `cwnd` = `cwnd`+1 is the additive increase and `cwnd` = `cwnd`/2 is the multiplicative decrease. Typically, setting `cwnd`=`cwnd`/2 is a medium-term goal; in fact, TCP Tahoe briefly sets `cwnd`=1 in the immediate aftermath of an actual timeout. With no losses, TCP will send successive windowfuls of, say, 20, 21, 22, 23, 24, ... This amounts to conservative "probing" of the network and, in particular, of the queue at the bottleneck router. TCP tries larger `cwnd` values because the absence of loss means the current `cwnd` is below the "network ceiling"; that is, the queue at the bottleneck router is not yet overfull.

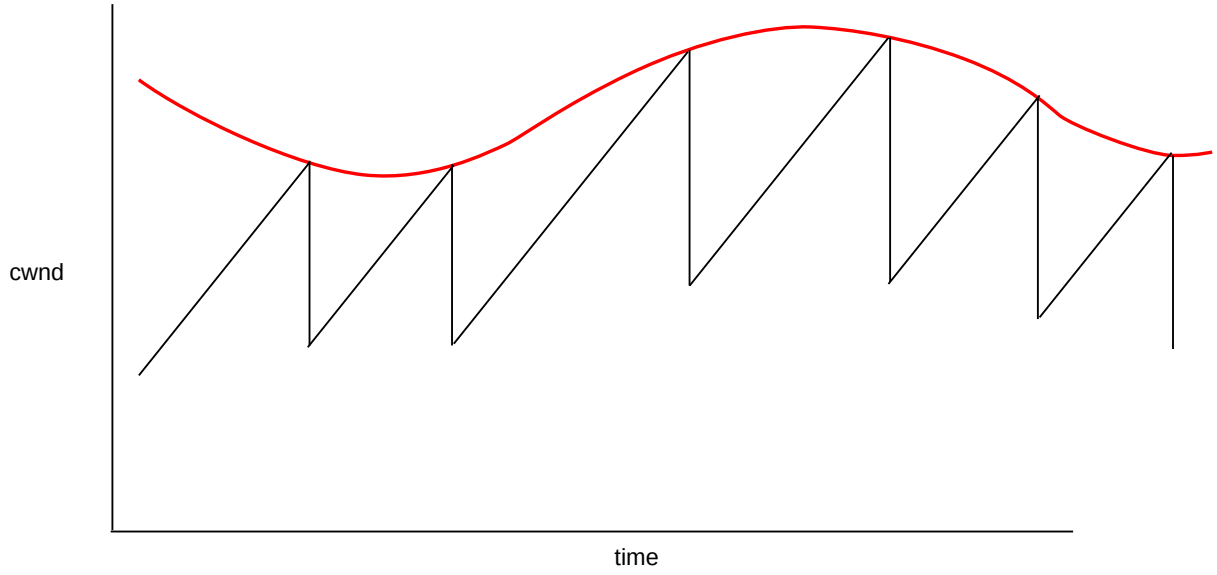
If a loss occurs (including multiple losses in a single windowful), TCP's response is to cut the window size in half. (As we will see, TCP Tahoe actually handles this in a somewhat roundabout way.) Informally, the idea is that the sender needs to respond aggressively to congestion. More precisely, lost packets mean the queue of the bottleneck router has filled, and the sender needs to dial back to a level that will allow the queue to clear. If we assume that the transit capacity is roughly equal to the queue capacity (say each is equal to N), then we overflow the queue and drop packets when `cwnd` = 2N, and so `cwnd` = `cwnd`/2 leaves us with `cwnd` = N, which just fills the transit capacity and leaves the queue empty. (When the sender sets `cwnd`=N, the actual number of packets in transit takes at least one RTT to fall from 2N to N.)

Of course, assuming any relationship between transit capacity and queue capacity is highly speculative. On a 5,000 km fiber-optic link with a bandwidth of 10 Gbps, the round-trip transit capacity would be about 60 MB, or 60,000 1kB packets. Most routers probably do not have queues that large. Queue capacities in excess of the transit capacity are common, however. On the other hand, a competing model of a long-haul high-bandwidth TCP path is that the queue size should be a small fraction of the bandwidth×delay product. We return to this in *19.7 TCP and Bottleneck Link Utilization* and *21.5.1 Bufferbloat*.

Note that if TCP experiences a packet loss, and there is an actual timeout (as opposed to a packet loss detected by Fast Retransmit, *19.3 TCP Tahoe and Fast Retransmit*), then the sliding-window pipe has drained. No packets are in flight. No self-clocking can govern new transmissions. Sliding windows therefore needs to restart from scratch.

The congestion-avoidance algorithm leads to the classic "TCP sawtooth" graph, where the peaks are at the points where the slowly rising `cwnd` crossed above the "network ceiling". We emphasize that the

TCP sawtooth is specific to TCP Reno and related TCP implementations that share Reno's additive-increase/multiplicative-decrease mechanism.



TCP Sawtooth, red curve represents the network capacity

During periods of no loss, TCP's `cwnd` increases linearly; when a loss occurs, TCP sets `cwnd` = `cwnd`/2. This diagram is an idealization as when a loss occurs it takes the sender some time to discover it, perhaps as much as the TimeOut interval.

The fluctuation shown here in the red ceiling curve is somewhat arbitrary. If there are only one or two other competing senders, the ceiling variation may be quite dramatic, but with many concurrent senders the variations may be smoothed out.

For some TCP sawtooth graphs created through actual simulation, see *31.2.1 Graph of cwnd v time* and *31.4.1 Some TCP Reno cwnd graphs*.

19.1.1.1 A first look at fairness

The transit capacity of the path is more-or-less unvarying, as is the physical capacity of the queue at the bottleneck router. However, these capacities are also shared with other connections, which may come and go with time. This is why the ceiling does vary in real terms. If two other connections share a path with total capacity 60 packets, the "fairest" allocation might be for each connection to get about 20 packets as its share. If one of those other connections terminates, the two remaining ones might each rise to 30 packets. And if instead a fourth connection joins the mix, then after equilibrium is reached each connection might hope for a fair share of 15 packets.

Will this kind of "fair" allocation actually happen? Or might we end up with one connection getting 90% of the bandwidth while two others each get 5%?

Chiu and Jain [CJ89] showed that the additive-increase/multiplicative-decrease algorithm does indeed converge to roughly equal bandwidth sharing when two connections have a common bottleneck link, provided also that

- both connections have the same RTT
- during any given RTT, either both connections experience a packet loss, or neither connection does

To see this, let `cwnd1` and `cwnd2` be the connections' congestion-window sizes, and consider the quantity `cwnd1 – cwnd2`. For any RTT in which there is no loss, `cwnd1` and `cwnd2` both increment by 1, and so `cwnd1 – cwnd2` stays the same. If there is a loss, then both are cut in half and so `cwnd1 – cwnd2` is also cut in half. Thus, over time, the original value of `cwnd1 – cwnd2` is repeatedly cut in half (during each RTT in which losses occur) until it dwindles to inconsequentiality, at which point `cwnd1` $\simeq$ `cwnd2`.

Graphical and tabular versions of this same argument are in the next chapter, in *20.3 TCP Reno Fairness with Synchronized Losses*.

The second bulleted hypothesis above we may call the **synchronized-loss hypothesis**. While it is very reasonable to suppose that the two connections will experience the same number of losses as a long-term *average*, it is a much stronger statement to suppose that all loss events are shared by both connections. This behavior may not occur in real life and has been the subject of some debate; see [GV02]. We return to this point in *31.3 Two TCP Senders Competing*. Fortunately, equal-RTT fairness still holds if each connection is *equally likely* to experience a packet loss: both connections will have the same loss rate, and so, as we shall see in *21.2 TCP Reno loss rate versus cwnd*, will have the same `cwnd`. However, convergence to fairness may take rather much longer. In *20.3 TCP Reno Fairness with Synchronized Losses* we also look at some alternative hypotheses for the unequal-RTT case.

19.2 Slow Start

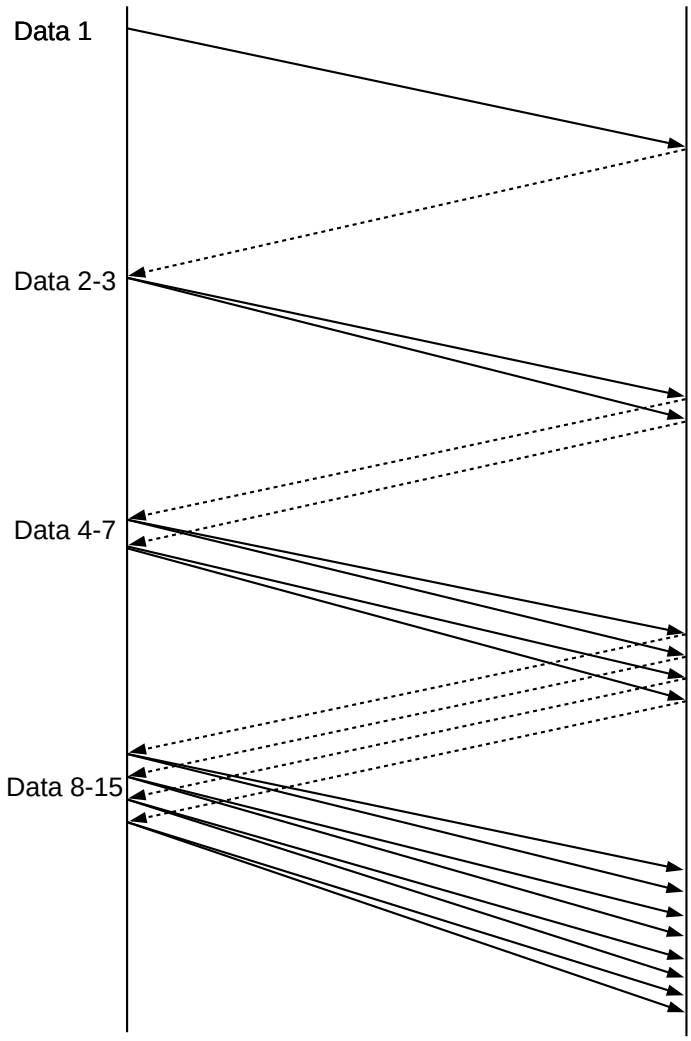
How do we make that initial guess as to the network capacity? What value of `cwnd` should we begin with? And even if we have a good target for `cwnd`, how do we avoid flooding the network sending an initial burst of packets?

The TCP Reno answer is known as **slow start**. If you are trying to guess a number in a fixed range, you are likely to use binary search. Not knowing the range for the "network ceiling", a good strategy is to guess `cwnd`=1 (or `cwnd`=2) at first and keep doubling until you have gone too far. Then revert to the previous guess, which is known to have worked. At this point you are guaranteed to be within 50% of the true capacity.

The actual slow-start mechanism is to increment `cwnd` by 1 for each ACK received. This seems linear, but that is misleading: after we send a windowful of packets (`cwnd` many), we have received `cwnd` ACKs and so have incremented `cwnd`-many times, and so have set `cwnd` to (`cwnd`+`cwnd`) = 2×`cwnd`. In other words, `cwnd`=`cwnd`×2 after each *windowful* is the same as `cwnd`+=1 after each *packet*.

Assuming packets travel together in windowfuls, all this means `cwnd` *doubles* each RTT during slow start; this is possibly the only place in the computer science literature where exponential growth is described as "slow". It is indeed slower, however, than the alternative of sending an entire windowful at once.

Here is a diagram of slow start in action. This diagram makes the implicit assumption that the no-load RTT is large enough to hold well more than the 8 packets of the maximum window size shown.



Slow Start with discrete packet flights

For a different case, with a much smaller RTT, see *19.2.3 Slow-Start Multiple Drop Example*.

Eventually the bottleneck queue gets full, and drops a packet. Let us suppose this is after N RTTs, so `cwnd`=2^N. Then during the previous RTT, `cwnd`=2^{N-1} worked successfully, so we go back to that previous value by setting `cwnd` = `cwnd`/2.

19.2.1 TCP Reno Per-ACK Responses

During slow start, incrementing `cwnd` by one per ACK received is equivalent to doubling `cwnd` after each windowful. We can find a similar equivalence for the congestion-avoidance phase, above.

During congestion avoidance, `cwnd` is incremented by 1 after each windowful. To formulate this as a **per-ACK** increase, we spread this increment of 1 over the entire windowful, which of course has size `cwnd`. This amounts to the following upon each ACK received:

`cwnd = cwnd + 1/cwnd`

This is a slight approximation, because `cwnd` keeps changing, but it works well in practice. Because TCP actually measures `cwnd` in bytes, floating-point arithmetic is normally not required; see exercise 14.0. An exact equivalent to the per-windowful incrementing strategy is `cwnd = cwnd + 1/cwnd`$_0$, where `cwnd`$_0$ is the value of `cwnd` at the start of that particular windowful. Another, simpler, approach is to use `cwnd += 1/cwnd`, and to keep the fractional part recorded, but to use floor(`cwnd`) (the integer part of `cwnd`) when actually sending packets.

Most actual implementations keep track of `cwnd` in bytes, in which case using integer arithmetic is sufficient until `cwnd` becomes quite large.

If **delayed ACKs** are implemented (*18.8 TCP Delayed ACKs*), then in bulk transfers one arriving ACK actually acknowledges two packets. **RFC 3465** permits a TCP receiver to increment `cwnd` by 2/`cwnd` in that situation, which is the response consistent with incrementing `cwnd` by 1 upon receipt of enough ACKs to acknowledge an entire windowful.

19.2.2 Threshold Slow Start

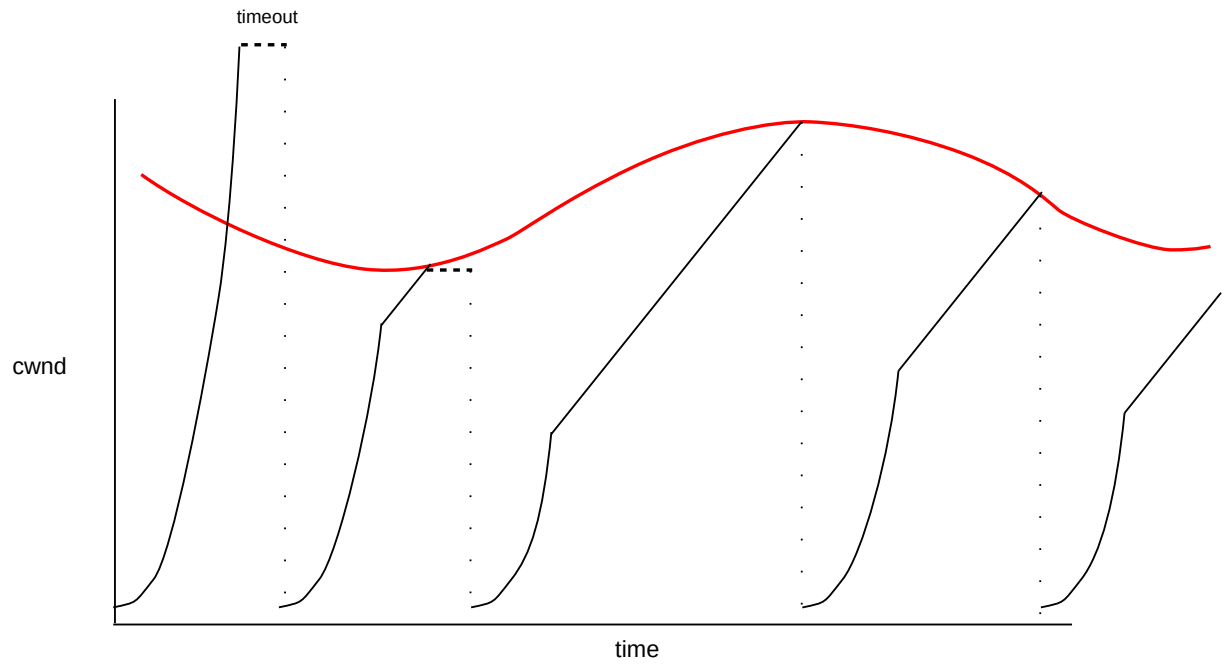
Sometimes TCP uses slow start even when it knows the working network capacity. After a packet loss and timeout, TCP knows that a new `cwnd` of `cwnd`$_{old}$/2 should work. If `cwnd` had been 100, TCP halves it to 50. The problem, however, is that after timeout there are no returning ACKs to self-clock the continuing transmission, and we do not want to dump 50 packets on the network all at once. So in restarting the flow TCP uses what might be called **threshold slow start**: it uses slow-start, but stops when `cwnd` reaches the target. Specifically, on packet loss we set the variable `ssthresh` to `cwnd`/2; this is our new target for `cwnd`. We set `cwnd` itself to 1, and switch to the slow-start mode (`cwnd += 1` for each ACK). However, as soon as `cwnd` reaches `ssthresh`, we switch to the congestion-avoidance mode (`cwnd += 1/cwnd` for each ACK). Note that the transition from threshold slow start to congestion avoidance is completely natural, and easy to implement.

TCP will use threshold slow-start whenever it is restarting from a pipe drain; that is, every time slow-start is needed after its very first use. (If a connection has simply been *idle*, non-threshold slow start is typically used when traffic starts up again.)

Threshold slow-start can be seen as an attempt at combining rapid window expansion with self-clocking.

By comparison, we might refer to the initial, non-threshold slow start as **unbounded slow start**. Note that unbounded slow start serves a fundamentally different purpose – initial probing to determine the network ceiling to within 50% – than threshold slow start.

Here is the TCP sawtooth diagram above, modified to show timeouts and slow start. The first two packet losses are displayed as "coarse timeouts"; the rest are displayed as if Fast Retransmit, below, were used.



TCP Tahoe Sawtooth, red curve represents the network capacity
Slow Start is used after each packet loss until ssthresh is reached

RFC 2581 allows slow start to begin with `cwnd`=2.

19.2.3 Slow-Start Multiple Drop Example

Slow start has the potential to cause multiple dropped packets at the bottleneck link; packet losses continue for quite some time because the TCP sender is slow to discover them. The network topology is as follows, where the A–R link is infinitely fast and the R–B link has a bandwidth in the R⟶B direction of 1 packet/ms.

Assume that R has a queue capacity of 100, not including the packet it is currently forwarding to B, and that ACKs travel instantly from B back to A. In this and later examples we will continue to use the Data[N]/ACK[N] terminology of *8.2 Sliding Windows*, beginning with N=1; TCP numbering is not done quite this way but the distinction is inconsequential.

When A uses slow-start here, the successive windowfuls will almost immediately begin to overlap. A will send one packet at T=0; it will be delivered at T=1. The ACK will travel instantly to A, at which point A will send two packets. From this point on, ACKs will arrive regularly at A at a rate of one per second. Here is a brief chart:

Time	A receives	A sends	R sends	R's queue
0		Data[1]	Data[1]	
1	ACK[1]	Data[2],Data[3]	Data[2]	Data[3]
2	ACK[2]	4,5	3	4,5
3	ACK[3]	6,7	4	5..7
4	ACK[4]	8,9	5	6..9
5	ACK[5]	10,11	6	7..11
..				
N	ACK[N]	2N,2N+1	N+1	N+2 .. 2N+1

At T=N, R's queue contains N packets. At T=100, R's queue is full. Data[200], sent at T=100, will be delivered and acknowledged at T=200, giving it an RTT of 100. At T=101, R receives Data[202] and Data[203] and drops the latter one. Unfortunately, A's timeout interval must of course be greater than the RTT, and so A will not detect the loss until, at an absolute minimum, T=200. At that point, A has sent packets up through Data[401], and the 100 packets Data[203], Data[205], . . . , Data[401] have all been lost. In other words, at the point when A *first* receives the news of one lost packet, in fact at least 100 packets have already been lost.

Fortunately, unbounded slow start generally occurs only once per connection.

19.2.4 Summary of TCP so far

So far we have the following features:

- Unbounded slow start at the beginning
- Congestion avoidance with AIMD once some semblance of a steady state is reached
- Threshold slow start after each loss
- Each threshold slow start transitioning naturally to congestion avoidance

Here is a table expressing the slow-start and congestion-avoidance phases in terms of manipulating `cwnd`.

phase	`cwnd` change, loss	`cwnd` change, no loss	
	per window	per window	per ACK
slow start	`cwnd`/2	`cwnd` *= 2	`cwnd` += 1
cong avoid	`cwnd`/2	`cwnd` +=1	`cwnd` += 1/`cwnd`

Viewing `cwnd`

Linux users can view the current values of `cwnd` and `ssthresh`, as well as a host of other TCP statistics, using the command `ss --tcp --info`.

The problem TCP often faces, in both slow-start and congestion-avoidance phases, is that when a packet is lost the sender will not detect this until much later (at least until the bottleneck router's current queue has been sent); by then, it may be too late to avoid further losses.

19.2.5 The initial value of `cwnd`

So far we have been assuming that slow start begins with `cwnd` equal to one packet, the value proposed in [JK88]. But 1988 was quite a while ago, and the initial `cwnd` has been creeping up. A decade later, **RFC 2414** proposed setting the initial `cwnd` (used on startup and after a timeout) to between two and four packets, with a maximum size of 3×1460 (three packets if the maximum Ethernet packet size of 1500 bytes is used). For a while following, the most common value was two packets. Then **RFC 6928**, in 2013, proposed an "experimental" change to an initial value of up to ten packets, where each can be 1460 bytes. This ten-packet value is now in relatively common use.

This increase in the initial `cwnd` value has mostly to do with the steadily increasing capacity of the Internet, and the decreased likelihood that a single extra packet will make a material difference. Still, the basic behavior of slow start remains the same.

Non-Reno TCPs also implement something like slow start, though it may look slightly different. TCP BBR (*22.16 TCP BBR*), for example, has a STARTUP mode that serves the same function as TCP Reno slow start.

19.3 TCP Tahoe and Fast Retransmit

TCP Tahoe has one more important feature. Recall that TCP ACKs are cumulative; if packets 1 and 2 have been received and now Data[4] arrives, but not yet Data[3], all the receiver can (and must!) do is to send back another ACK[2]. Thus, from the sender's perspective, if we send packets 1,2,3,4,5,6 and get back ACK[1], ACK[2], ACK[2], ACK[2], ACK[2], we can infer two things:

- Data[3] got lost, which is why we are stuck on ACK[2]
- Data 4,5 and 6 probably *did* make it through, and triggered the three duplicate ACK[2]s (the three ACK[2]s following the first ACK[2]).

The **Fast Retransmit** strategy is to resend Data[N] when we have received three dupACKs for Data[N-1]; that is, four ACK[N-1]'s in all. Because this represents a packet loss, we also set `ssthresh` = `cwnd`/2, set `cwnd`=1, and begin the threshold-slow-start phase. The effect of this is typically to reduce the delay associated with the lost packet from that of a full timeout, typically $2 \times$RTT, to just a little over a single RTT. The lost packet is now discovered *before* the TCP pipeline has drained. However, at the end of the next RTT, when the ACK of the retransmitted packet will return, the TCP pipeline *will* have drained, hence the need for slow start.

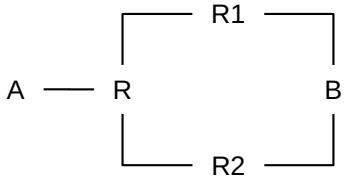
TCP Tahoe included all the features discussed so far: the `cwnd`+=1 and `cwnd`=`cwnd`/2 responses, slow start and Fast Retransmit.

Fast Retransmit waits for the *third* dupACK to allow for the possibility of moderate packet reordering. Suppose packets 1 through 6 are sent, but they arrive in the order 1,3,4,2,6,5, perhaps due to a router along the way with an architecture that is strongly parallelized. Then the ACKs that would be sent back would be as follows:

Received	Response
Data[1]	ACK[1]
Data[3]	ACK[1]
Data[4]	ACK[1]
Data[2]	ACK[4]
Data[6]	ACK[4]
Data[5]	ACK[6]

Waiting for the third dupACK is in most cases a successful compromise between responsiveness to lost packets and reasonable evidence that the data packet in question is actually lost.

However, a router that does more substantial delivery reordering would wreck havoc on connections using Fast Retransmit. In particular, consider the router R in the diagram below; when sending packets to B it might in principle wish to alternate on a packet-by-packet basis between the path via R1 and the path via R2. This would be a mistake; if the R1 and R2 paths had different propagation delays then this strategy would introduce major packet reordering. R should send all the packets belonging to any one TCP connection via a single path.



In the real world, routers generally go to considerable lengths to accommodate Fast Retransmit; in particular, use of multiple paths for a single TCP connection is almost universally frowned upon. Some actual data on packet reordering can be found in [VP97]; the author suggests that a switch to retransmission on the second dupACK would be risky.

19.4 TCP Reno and Fast Recovery

Fast Retransmit requires a sender to set `cwnd`=1 because the pipe has drained and there are no arriving ACKs to pace transmission. Fast Recovery is a technique that often allows the sender to avoid draining the pipe, and to move from `cwnd` to `cwnd`/2 in the space of a single RTT. TCP Reno is TCP Tahoe with the addition of Fast Recovery.

The idea is to use the arriving dupACKs to pace retransmission. We make the assumption that each arriving dupACK indicates that *some* data packet following the lost packet has been delivered successfully; it turns out not to matter which one. On discovery of the lost packet through Fast Retransmit, the goal is to set `cwnd`=`cwnd`/2; the next step is to figure out how many dupACKs we have to wait for before we can resume transmissions of new data.

Initially, at least, we assume that only one data packet is lost, though in the following section we will see that multiple losses can be handled via a slight modification of the Fast Recovery strategy.

During the recovery process, there is a problem with the direct use of sliding windows: the lost packet "pins" the lower end of the window until that packet is successfully retransmitted, so for the duration of the

recovery period the window cannot slide forward. The official specification of fast recovery, originally in **RFC 2001** and now in **RFC 5681**, describes retransmission in terms of `cwnd` **inflation** and **deflation**. If C is the value of `cwnd` at the time of the loss, then `cwnd` is steadily inflated to 1.5×C, allowing for the original window plus half a windowful more of new data. At the point the lost packet is successfully retransmitted, the window is "deflated" to `cwnd` = C/2; at this point the lower end of the window slides forward by C, and the upper end of the window does not move. This all works out, but can be a bit hard to follow.

Instead we will use the concept of Estimated FlightSize, or **EFS**, which is the sender's best guess at the number of outstanding packets. Under normal circumstances, EFS is the same as `cwnd`. The crucial Fast Recovery observation is that EFS should be decremented by 1 for each arriving dupACK, because the arrival of each dupACK means one less packet is in transit, and so transmission of new packets can resume when EFS is reduced to half of the original value of `cwnd`. Our EFS approach is equivalent to the inflationary approach at least when slow start is not involved.

Linux `cwnd` *is* Estimated FlightSize

This chapter defines `cwnd` to be the sender winsize strictly construed. As such, packet N+`cwnd` cannot be sent until packet N is ACKed. However, the Linux kernel actually uses `cwnd` as a synonym for Estimated FlightSize, which simplifies the Fast-Recovery code. This usage applies, in fact, to all TCP varieties, though it often makes little difference. See tcp_cwnd_test().

We first outline the general case, and then look at a specific example. Let `cwnd` = N, and suppose packet 1 is lost (packet numbers here may be taken as relative). Until packet 1 is retransmitted, the sender can only send up through packet N (Data[N] can be sent only after ACK[0] has arrived at the sender). The receiver will send N–1 dupACK[0]s representing packets 2 through N.

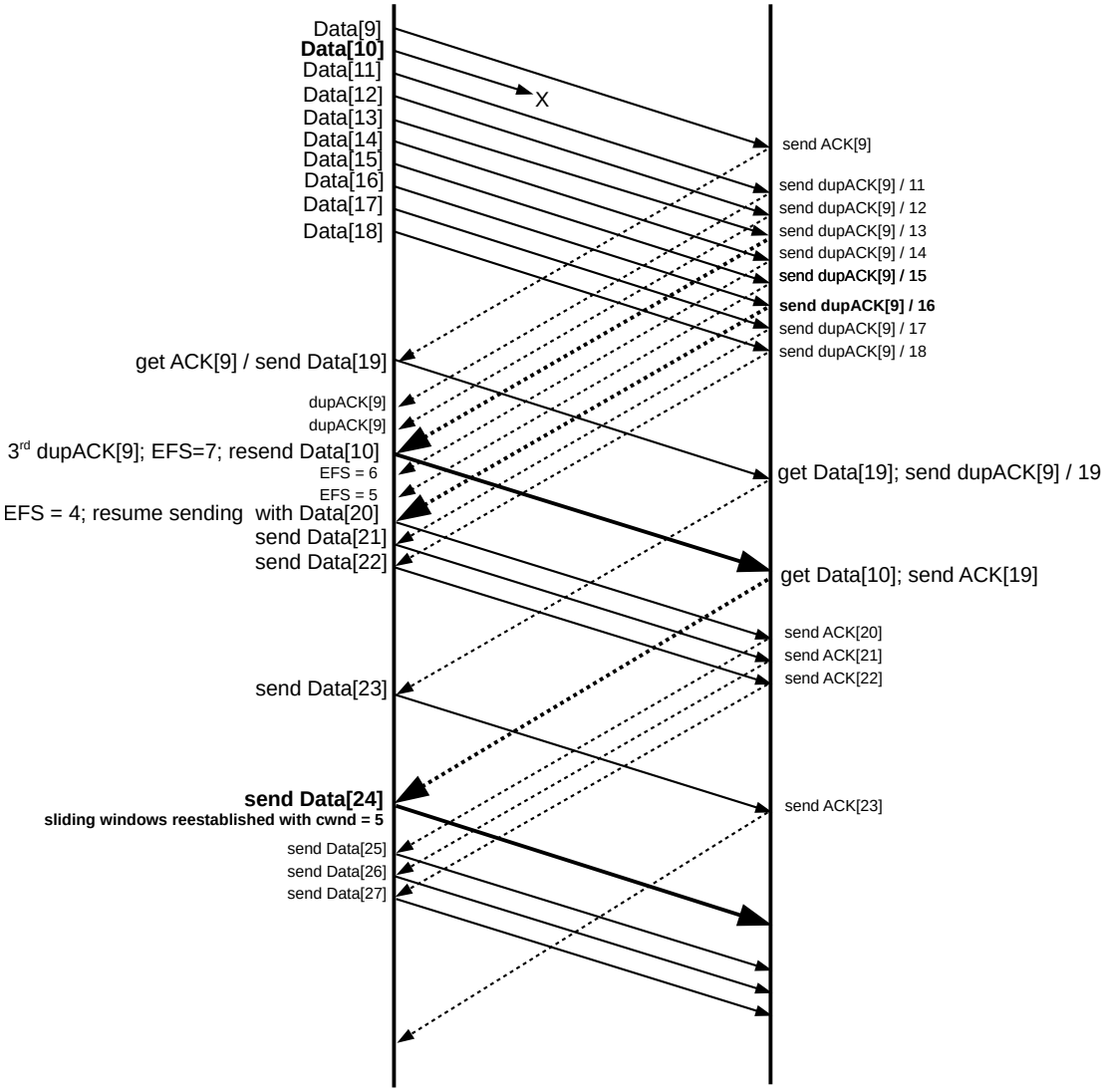
At the point of the third dupACK, when the loss of Data[1] is discovered, the sender calculates as follows: EFS had been `cwnd` = N. Three dupACKs have arrived, representing three later packets no longer in flight, so EFS is now N–3. At this point the sender realizes a packet has been lost, which makes EFS = N–4 briefly, but that packet is then immediately retransmitted, which brings EFS back to N–3.

The sender expects at this point to receive N–4 more dupACKs, followed by one new ACK for the retransmission of the packet that was lost. This last ACK will be for the entire original windowful.

The new target for `cwnd` is N/2 (for simplicity, we will assume N is even). So, we wait for N/2 – 3 more dupACKs to arrive, at which point EFS is N–3–(N/2–3) = N/2. *After* this point the sender will resume sending new packets; it will send one new packet for each of the N/2–1 subsequently arriving dupACKs (recall that there are N–1 dupACKs in all). These new transmissions will be Data[N+1] through Data[N+(N/2–1)].

After the last of the dupACKs will come the ACK corresponding to the retransmission of the lost packet; it will be ACK[N], acknowledging all of the original windowful. At this point, there are N/2 – 1 unacknowledged packets Data[N+1] through Data[N+(N/2)–1]. The sender now sends Data[N+N/2] and is thereby able to resume sliding windows with `cwnd` = N/2: the sender has received ACK[N] and has exactly one full windowful outstanding for the new value N/2 of `cwnd`. That is, we are right where we are supposed to be.

Here is a diagram illustrating Fast Recovery for `cwnd`=10. Data[10] is lost.



Data[9] elicits the initial ACK[9], and the nine packets Data[11] through Data[19] each elicit a dupACK[9]. We denote the dupACK[9] elicited by Data[N] by dupACK[9]/N; these are shown along the upper right. Unless SACK TCP (below) is used, the sender will have no way to determine N or to tell these dupACKs apart. When dupACK[9]/13 (the third dupACK) arrives at the sender, the sender uses Fast Recovery to infer that Data[10] was lost and retransmits it. At this point EFS = 7: the sender has sent the original batch of 10 data packets, plus Data[19], and received one ACK and three dupACKs, for a total of 10+1-1-3 = 7. The sender has also inferred that Data[10] is lost (EFS –= 1) but then retransmitted it (EFS += 1). Six more dupACK[9]'s are on the way.

EFS is decremented for each subsequent dupACK arrival; after we get two more dupACK[9]'s, EFS is 5. The next dupACK[9] (dupACK[9]/16) reduces EFS to 4 and so allows us transmit Data[20] (which promptly bumps EFS back up to 5). We have

receive	send
dupACK[9]/16	Data[20]
dupACK[9]/17	Data[21]
dupACK[9]/18	Data[22]
dupACK[9]/19	Data[23]

We emphasize again that the TCP sender does not see the numbers 16 through 19 in the receive column above; it determines when to begin transmission by counting dupACK[9] arrivals.

Working Backwards

Figuring out when a fast-recovery sender should resume transmissions of new data is error-prone. Perhaps the simplest approach is to work backwards from the retransmitted lost packet: it should trigger at the receiver an ACK for the entire original windowful. When Data[10] above was lost, the "stuck" window was Data[10]-Data[19]. The retransmitted Data[10] thus triggers ACK[19]; when ACK[19] arrives, `cwnd` should be 10/2 = 5 so Data[24] should be sent. That in turn means the four packets Data[20] through Data[23] must have been sent earlier, via Fast Recovery. There are 10–1 = 9 dupACKs, so to send on the last four we must start with the sixth. The diagram above indeed shows new Fast Recovery transmissions beginning with the sixth dupACK.

The next thing to arrive at the sender side is the ACK[19] elicited by the retransmitted Data[10]; at the point Data[10] arrives at the receiver, Data[11] through Data[19] have already arrived and so the cumulative-ACK response is ACK[19]. The sender responds to ACK[19] with Data[24], and the transition to `cwnd`=5 is now complete.

During sliding windows without losses, a sender will send `cwnd` packets per RTT. If a "coarse" timeout occurs, typically it is not discovered until after at least one complete RTT of link idleness; there are additional underutilized RTTs during the slow-start phase. It is worth examining the Fast Recovery sequence shown in the illustration from the perspective of underutilized bandwidth. The diagram shows three round-trip times, as seen from the sender side. During the first RTT, the ten packets Data[9]-Data[18] are sent. The second RTT begins with the sending of Data[19] and continues through sending Data[22], along with the retransmitted Data[10]. The third RTT begins with sending Data[23], and includes through Data[27]. In terms of recovery efficiency, the RTTs send 9, 5 and 5 packets respectively (we have counted Data[10] twice); this is remarkably close to the ideal of reducing `cwnd` to 5 instantaneously.

Using the fast-recovery description in terms of inflation from **RFC 5681**, inflation would begin at the point the sender resumed transmitting new packets, at which point `cwnd` would be incremented for each dupACK. In the diagram above, at the instant labeled "send Data[24]" on the sender side, `cwnd` would momentarily become 15, representing the window Data[10]..Data[24]. As soon as the sender realized that the lost packet Data[10] had been acknowledged, via ACK[19], `cwnd` would immediately deflate to 5, representing the window Data[20]..Data[24]. For a diagram illustrating `cwnd` inflation and deflation, see *32.2.1 Running the Script*.

There is one more addition to Fast Recovery, described in **RFC 3042**, and known as **Limited Transmit**. As described above, transmission of new data begins with the third dupACK, which is the point at which Fast Recovery is initiated. Limited Transmit means that one new data packet is also transmitted for each of the first two dupACKs, on a "just in case" basis. These two transmissions are "borrowed" against future

forward motion of the send window; the value of `cwnd` is not changed and the first "skipped" packet is not retransmitted. If there was no packet loss, and the dupACKs simply resulted from packet reordering, no harm is done. Otherwise, Fast Retransmit gets a slightly earlier start. For small window sizes this can make a significant difference.

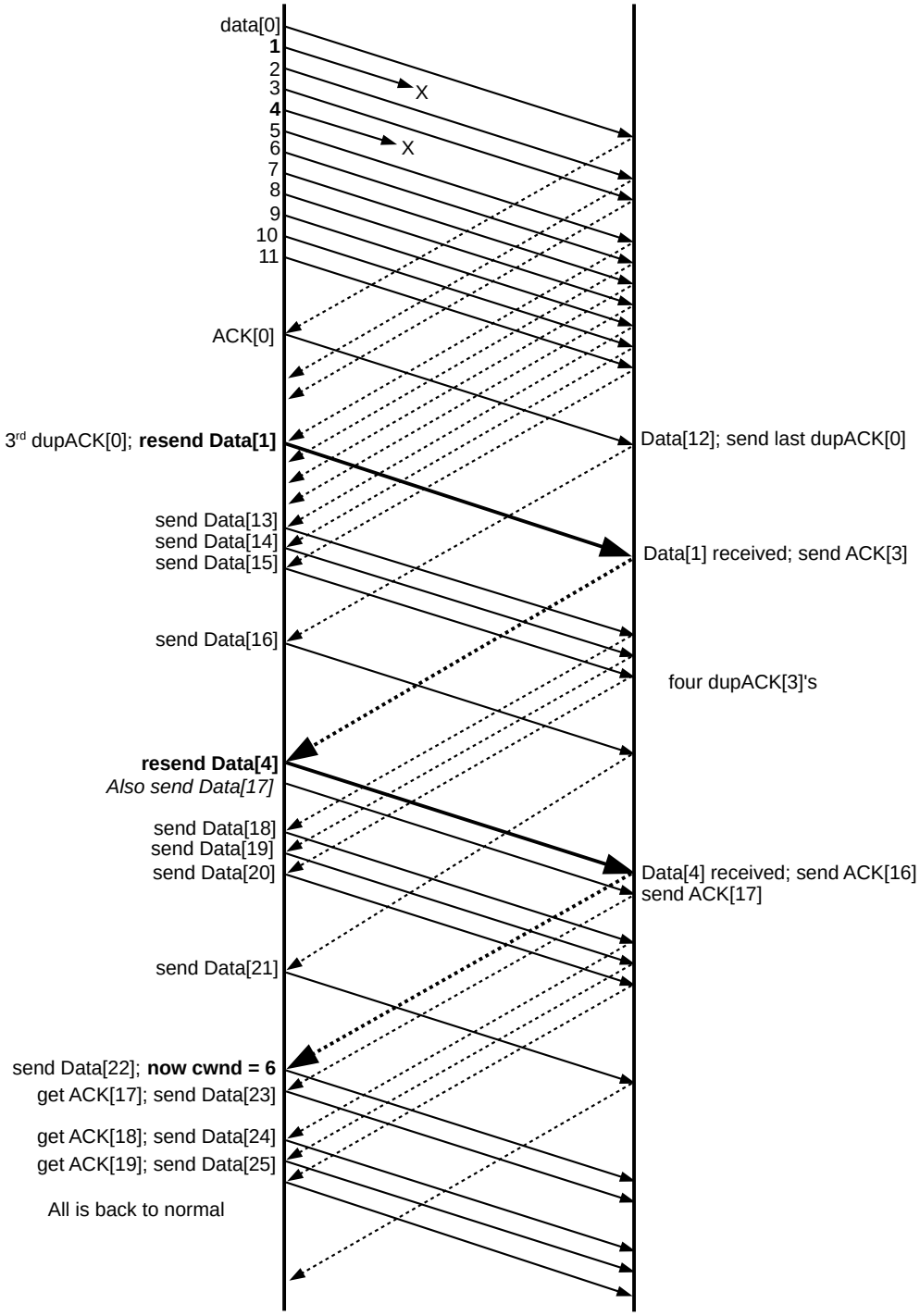
19.5 TCP NewReno

TCP NewReno, described in [JH96] and **RFC 2582** (currently **RFC 6582**), is a modest tweak to Fast Recovery which greatly improves handling of the case when two or more packets are lost in a windowful. It is generally considered to be a part of contemporary TCP Reno. We again describe it in terms of Estimated FlightSize rather than in terms of `cwnd` inflation and deflation.

If two data packets are lost and the first is retransmitted, the receiver will acknowledge data up to just before the second packet, and then continue sending dupACKs of this until the second lost packet is also retransmitted. These ACKs of data up to just before the second packet are sometimes called **partial ACKs**, because retransmission of the first lost packet did not result in an ACK of all the outstanding data. The NewReno mechanism uses these partial ACKs as evidence to retransmit later lost packets, and also to keep pacing the Fast Recovery process.

In the diagram below, packets 1 and 4 are lost in a window 0..11 of size 12. Initially the sender will get dupACK[0]'s; the first 11 ACKs (dashed lines from right to left) are ACK[0] and 10 dupACK[0]'s. When packet 1 is successfully retransmitted on receipt of the third dupACK[0], the receiver's response will be ACK[3] (the heavy dashed line). This is the first partial ACK (a full ACK would have been ACK[12]). On receipt of any partial ACK during the Fast Recovery process, TCP NewReno assumes that the immediately following data packet was lost and retransmits it immediately; the sender does not wait for three dupACKs because if the following data packet had not been lost, no instances of the partial ACK would ever have been generated, even if packet reordering had occurred.

The TCP NewReno sender response here is, in effect, to treat each partial ACK as a dupACK[0], except that the sender *also* retransmits the data packet that – based upon receipt of the partial ACK – it is able to infer is lost. NewReno continues pacing Fast Recovery by whatever ACKs arrive, whether they are the original dupACKs or later partial ACKs or dupACKs.



When the receiver's first ACK[3] arrives at the sender, NewReno infers that Data[4] was lost and resends it; this is the second heavy data line. No dupACK[3]'s need arrive; as mentioned above, the sender can infer from the single ACK[3] that Data[4] is lost. The sender also responds as if another dupACK[0] had arrived, and sends Data[17].

The arrival of ACK[3] signals a reduction in the EFS by 2: one for the inference that Data[4] was lost, and

one as if another dupACK[0] had arrived; the two transmissions in response (of Data[4] and Data[17]) bring EFS back to where it was. At the point when Data[16] is sent the actual (not estimated) flightsize is 5, not 6, because there is one less dupACK[0] due to the loss of Data[4]. However, once NewReno resends Data[4] and then sends Data[17], the actual flightsize is back up to 6.

There are four more dupACK[3]'s that arrive. NewReno keeps sending new data on receipt of each of these; these are Data[18] through Data[21].

The receiver's response to the retransmitted Data[4] is to send ACK[16]; this is the cumulative of all the data received up to that moment. At the point this ACK arrives back at the sender, it had just sent Data[21] in response to the fourth dupACK[3]; its response to ACK[16] is to send the next data packet, Data[22]. *The sender is now back to normal sliding windows*, with a `cwnd` of 6. Similarly, the Data[17] immediately following the retransmitted Data[4] elicits an ACK[17] (this is the first Data[N] to elicit an exactly matching ACK[N] since the losses began), and the corresponding response to the ACK[17] is to continue with Data[23].

As with the previous Fast Recovery example, we consider the number of packets sent per RTT; the diagram shows four RTTs as seen from the sender side.

RTT	First packet	Packets sent	count
first	Data[0]	Data[0]-Data[11]	12
second	Data[12]	Data[12]-Data[15], Data[1]	5
third	Data[16]	Data[16]-Data[20], Data[4]	6
fourth	Data[21]	Data[21]-Data[26]	6

Again, after the loss is detected we segue to the new `cwnd` of 6 with only a single missed packet (in the second RTT). NewReno is, however, only able to send one retransmitted packet per RTT.

Note that TCP Newreno, like TCPs Tahoe and Reno, is a **sender-side** innovation; the receiver does not have to do anything special. The next TCP flavor, SACK TCP, requires receiver-side modification.

19.6 Selective Acknowledgments (SACK)

A traditional TCP ACK is a cumulative acknowledgment of all data received up to that point. If Data[1002] is received but not Data[1001], then all the receiver can send is a duplicate ACK[1000]. This does indicate that *something* following Data[1001] made it through, but nothing more.

To provide greater specificity, TCP now provides a **Selective ACK** (SACK) option, implemented at the receiver. If this is available, the sender does not have to guess from dupACKs what has gotten through. The receiver can send an ACK that says:

- All packets up through 1000 have been received (the cumulative ACK)
- All packets up through 1050 have been received *except for* 1001, 1022, and 1035.

The second line is the SACK part. Almost all TCP implementations now support this.

Specifically, SACKs include the following information; the additional data beyond the cumulative ACK is included in a TCP Option field.

- The latest cumulative ACK

- The *three* most recent blocks of consecutive packets received

Thus, if we have lost 1001, 1022, 1035, and now 1051, and the highest received is 1060, the SACK might say:

- All packets up through 1000 have been received
- 1060-1052 have been received
- 1050-1036 have been received
- 1034-1023 have been received

From this the sender know 1001s and 1022 were not received, but nothing about the packets in between. However, if the sender has been paying close attention to the previous SACKs received, it likely already knows that all packets 1002 through 1021 have been received.

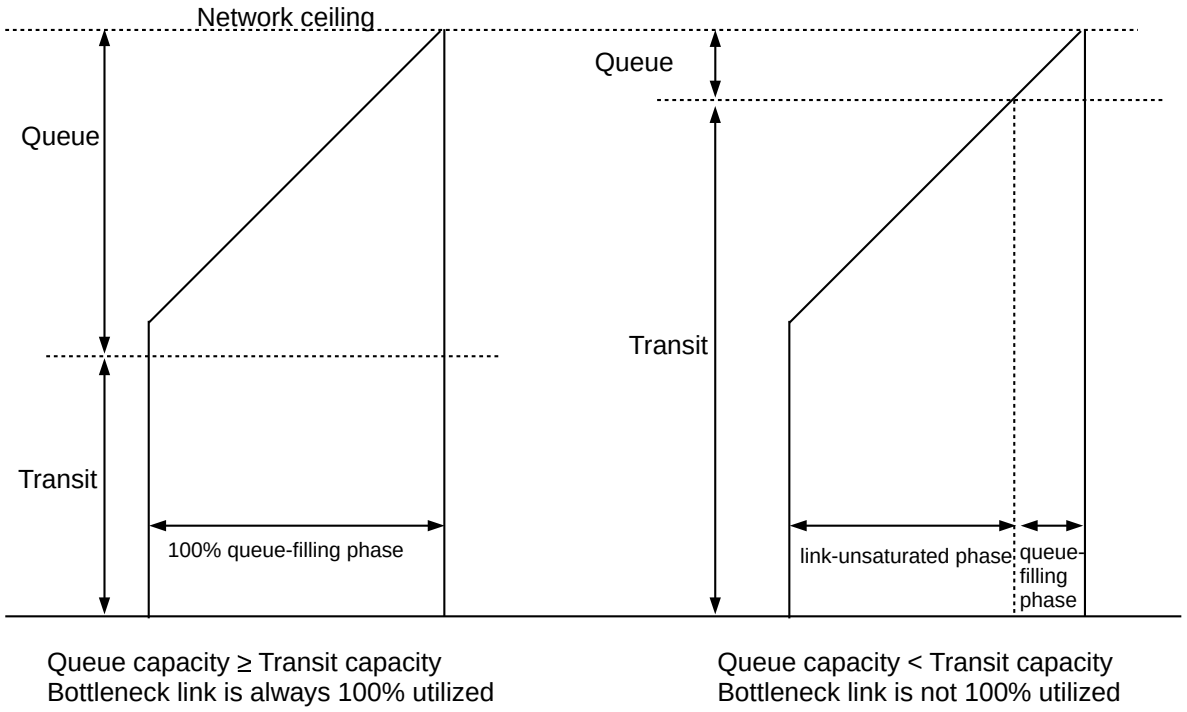
The term **SACK TCP** is typically used to mean that the receiving side supports selective ACKs, and the sending side is a straightforward modification of TCP Reno to take advantage of them.

In practice, selective ACKs provide at best a modest performance improvement in many situations; TCP NewReno does rather well, in moderate-loss environments. The paper [FF96] compares Tahoe, Reno, NewReno and SACK TCP, in situations involving from one to four packet losses in a single RTT. While Classic Reno performed poorly with two packet losses in a single RTT and extremely poorly with three losses, the three-loss NewReno and SACK TCP scenarios played out remarkably similarly. Only when connections experienced four losses in a single RTT did SACK TCP's performance start to pull slightly ahead of that of NewReno.

19.7 TCP and Bottleneck Link Utilization

Consider a TCP Reno sender with no competing traffic. As `cwnd` saws up and down, what happens to throughput? Do those halvings of `cwnd` result in at least a dip in throughput? The answer depends to some extent on the size of the queue ahead of the bottleneck link, relative to the transit capacity of the path. As was discussed in *8.3.2 RTT Calculations*, when `cwnd` is less than the transit capacity, the link is less than 100% utilized and the queue is empty. When `cwnd` is more than the transit capacity, the link is saturated (100% utilized) and the queue has about (`cwnd` – transit_capacity) packets in it.

The diagram below shows two TCP Reno teeth; in the first, the queue capacity exceeds the path transit capacity and in the second the queue capacity is a much smaller fraction of the total.



In the first diagram, the bottleneck link is always 100% utilized, even at the left edge of the teeth. In the second the interval between loss events (the left and right margins of the tooth) is divided into a **link-unsaturated** phase and a **queue-filling phase**. In the unsaturated phase, the bottleneck link utilization is less than 100% and the queue is empty; in the later phase, the link is saturated and the queue begins to fill.

Consider again the idealized network below, with an R–B bandwidth of 1 packet/ms.

We first consider the queue ⩾ transit case. Assume that the total $\text{RTT}_{\text{noLoad}}$ delay is 100 ms, mostly due to propagation delay; this makes the bandwidth×delay product 100 packets. The question for consideration is to what extent TCP Reno, once slow-start is over, sometimes leaves the R–B link idle.

The R–B link will be saturated at all times provided A always keeps 100 packets in transit, that is, we always have `cwnd` ⩾ 100 (*8.3.2 RTT Calculations*). If $\texttt{cwnd}_{\text{min}}$ = 100, then $\texttt{cwnd}_{\text{max}} = 2\times\texttt{cwnd}_{\text{min}}$ = 200. For this to be the maximum, the queue capacity must be at least 99, so that the path can accommodate 199 packets without loss: 100 packets in transit plus 99 packets in the queue. In general, TCP Reno never leaves the bottleneck link idle as long as the queue capacity in front of that link is at least as large as the path round-trip transit capacity.

Now suppose instead that the queue size is 49, or about 50% of the transit capacity. Packet loss will occur when `cwnd` reaches 150, and so $\texttt{cwnd}_{\text{min}}$ = 75. Qualitatively this case is represented by the second diagram above, though the queue-to-network_ceiling proportion illustrated there is more like 1:8 than 1:3. There are now periods when the R–B link is idle. During RTT intervals when `cwnd`=75, throughput will be 75% of the maximum and the R–B link will be idle 25% of the time.

However, `cwnd` will be 75 just for the first RTT following the loss. After 25 RTTs, `cwnd` will be back up to 100, and the link will be saturated. So we have 25 RTTs with an average `cwnd` of 87.5 (=&75+100)/2), meaning the link is 87.5% saturated, followed by 50 RTTs where the link is 100% saturated. The long-term average here is 95.8% utilization of the bottleneck link. This is not bad at all, given that using 10% of the link bandwidth on packet headers is almost universally considered reasonable. Furthermore, at the point when `cwnd` drops after a loss to $\texttt{cwnd}_{min}$=75, the queue must have been full. It may take one or two RTTs for the queue to drain; during this time, link utilization will be even higher.

If most or all of the time the bottleneck link is saturated, as in the first diagram, it may help to consider the average queue size. Let the queue capacity be C_{queue} and the transit capacity be $C_{transit}$, with $C_{queue} > C_{transit}$. Then `cwnd` will vary from a maximum of C_{queue}+$C_{transit}$ to a minimum of what works out to be (C_{queue}-$C_{transit}$)/2 + $C_{transit}$. We would expect an average queue size about halfway between these, less the $C_{transit}$ term: 3/4×C_{queue} - 1/4×$C_{transit}$. If C_{queue}=$C_{transit}$, the expected average queue size should be about C_{queue}/2.

See exercises 12.0 and 12.5.

19.7.1 TCP Queue Sizes

From the perspective of link utilization, the previous section suggests that router queues be larger rather than smaller. A queue capacity at least as large as transit capacity seems like an excellent choice. To configure a router this way, we first make an educated guess at the average RTT, and then multiply this by the output bandwidth to get the desired queue capacity. For an average RTT of 50 ms, a bandwidth of 1 Gbps leads to a queue capacity of about 6 MB, or 4000 packets of 1500 bytes each. If the numbers rise to 100 ms and 10 Gbps, queue capacity needs to be 125 MB.

Unfortunately, while large queues are helpful when the traffic consists exclusively of bulk TCP transfers, they introduce proportionately large queuing delays that can wreak havoc on real-time traffic. A bottleneck router with a queue size matching a flow's bandwidth×delay product will *double* the RTT for that flow, at points when the queue is full. Worse, if the goal is 100% TCP link utilization always, then the router queue must be sized for the highest-bandwidth flow with the longest RTT; shorter TCP connections will encounter a queue much larger than necessary. This problem of large queue capacity leading to excessive delay is known as **bufferbloat**; we will return to it at *21.5.1 Bufferbloat*.

Because of the delay problems brought on by large queues, TCP connections must sometimes pass through bottleneck routers with small queues. In this case a tooth of a TCP Reno connection is divided into a large link-unsaturated phase and a small queue-filling phase.

The *need* for large buffers, if near-100% queue utilization is the goal, is to a large degree specific to the TCP Reno sawtooth. Some other TCP implementations (in particular TCP Vegas, *22.6 TCP Vegas*), do not overfill the queue. However, TCP Vegas does not compete well with TCP Reno, at least with traditional FIFO queuing (*20.1 A First Look At Queuing*) (but see *23.6.1 Fair Queuing and Bufferbloat*).

The worst case for TCP link utilization is if the queue size is close to zero. Using again a bandwidth×delay product 100 of packets, a zero-sized queue will mean that $\texttt{cwnd}_{max}$ will be 100 (or 101), and so $\texttt{cwnd}_{min}$ will be 50. Link utilization therefore ranges, over the lifetime of the tooth, from a low of 50/100 = 50% to a high of 100%; the average utilization is **75%**. While this is not ideal, and while some non-Reno TCP variants have attempted to improve this figure, 75% link utilization is not all that bad, and can be compared with the 10% of the bandwidth consumed as packet headers (though that figure assumes 512 bytes of data

per packet, which is low). (A literally zero-sized queue will not work at all well; one reason – though not the only one – is that TCP Reno sends a two-packet burst whenever `cwnd` is incremented.)

Traffic mix has a major influence on the appropriate queue size. For example, the analysis of the previous section assumed a single long-term TCP connection. The link-utilization situation improves with increasing numbers of TCP connections, at least if the losses are unsynchronized, because the halving of one connection's `cwnd` has a proportionately smaller impact on the total queue use. In [AKM04] it is shown that for a router with N TCP connections with unsynchronized losses, a queue size of ($RTT_{average}$ × bandwidth)/$\sqrt{N}$ is sufficient to keep the link almost always saturated. Larger values of N here are typically associated with "core" (backbone) routers. The paper [EGMR05] proposes even smaller buffer capacities, on the order of the logarithm of the maximum window size. The argument makes two important assumptions, however: first, that we are willing to tolerate a link utilization somewhat less than 100% (though greater than 75%), and second, perhaps more importantly, that TCP is modified so as to spread out any packet bursts – even bursts of size two – over small intervals of time.

There are other problems created by too-small queues, even if we are willing to accept 75% link utilization. Internet traffic, not unlike city-bus traffic, tends to "bunch up"; queues serve as a way to keep these packet bunches from leading to unnecessary losses. For one example of unexpected traffic bunching, see *31.4.1.3 Transient queue peaks*. Increased traffic randomization helps reduce the need for very large queues, but may increase the bunching effect. Internet "core" routers see more highly randomized traffic than end-user or "edge" routers; queues in the latter are often the most difficult to configure.

We will return to the issue of link utilization in *31.2.6 Single-sender Throughput Experiments* and (for two senders) *31.3.10.2 Higher bandwidth and link utilization*, using the ns simulator to get experimental data. See also exercise 12.0.

Finally, the queue capacity does not necessarily have to remain static. We will return to this point in *21.5 Active Queue Management*. Furthermore, many queue-size problems ultimately spring from the fact that all traffic is being dumped into a single FIFO queue; we will look at alternative queuing strategies in *23 Queuing and Scheduling*. For a particular example related to bufferbloat, see *23.6.1 Fair Queuing and Bufferbloat*.

19.8 Single Packet Losses

Again assuming no competition on the bottleneck link, the TCP Reno additive-increase policy has a simple consequence: at the end of each tooth, only a single packet will be lost.

To see this, let A be the sender, R be the bottleneck router, and B be the receiver:

Let T be the bandwidth delay at R, so that packets leaving R are spaced at least time T apart. A will therefore transmit packets T time units apart, except for those times when `cwnd` has just been incremented and A sends a pair of packets back-to-back. Let us call the second packet of such a back-to-back pair the "extra" packet. To simplify the argument slightly, we will assume that the two packets of a pair arrive at R essentially simultaneously.

Only an extra packet can result in an increase in queue utilization; every other packet arrives after an interval T from the previous packet, giving R enough time to remove a packet from its queue.

A consequence of this is that `cwnd` will reach the sum of the transit capacity and the queue capacity without R dropping a packet. (This is not necessarily the case if a `cwnd` this large were sent as a single burst.)

Let C be this combined capacity, and assume `cwnd` has reached C. When A executes its next `cwnd` += 1 additive increase, it will as usual send a pair of back-to-back packets. The second of this pair – the extra – is doomed; it will be dropped when it reaches the bottleneck router.

At this point there are C = `cwnd` – 1 packets outstanding, all spaced at time intervals of T. Sliding windows will continue normally until the ACK of the packet just before the lost packet arrives back at A. After this point, A will receive only dupACKs. A has received C = `cwnd`–1 ACKs since the last increment to `cwnd`, but must receive C+1 = `cwnd` ACKs in order to increment `cwnd` again. This will not happen, as no more new ACKs will arrive until the lost packet is transmitted.

Following this, `cwnd` is reduced and the next sawtooth begins; the only packet that is lost is the "extra" packet of the previous flight.

See *31.2.3 Single Losses* for experimental confirmation, and exercise 15.0.

19.9 TCP Assumptions and Scalability

In the TCP design portrayed above, several embedded assumptions have been made. Perhaps the most important is that **every loss is treated as evidence of congestion**. As we shall see in the next chapter, this fails for high-bandwidth TCP (when rare random losses become significant); it also fails for TCP over wireless (either Wi-Fi or other), where lost packets are much more common than over Ethernet. See *21.6 The High-Bandwidth TCP Problem* and *21.7 The Lossy-Link TCP Problem*.

The TCP `cwnd`-increment strategy – to increment `cwnd` by 1 for each RTT – has some assumptions of scale. This mechanism works well for cross-continent RTT's on the order of 100 ms, and for `cwnd` in the low hundreds. But if `cwnd` = 2000, then it takes 100 RTTs – perhaps 20 seconds – for `cwnd` to grow 10%; linear increase becomes *proportionally* quite slow. Also, if the RTT is very long, the `cwnd` increase is slow. The absolute set-by-the-speed-of-light minimum RTT for geosynchronous-satellite Internet is 480 ms, and typical satellite-Internet RTTs are close to 1000 ms. Such long RTTs also lead to slow `cwnd` growth; furthermore, as we shall see below, such long RTTs mean that these TCP connections compete poorly with other connections. See *21.8 The Satellite-Link TCP Problem*.

Another implicit assumption is that if we have a lot of data to transfer, we will send all of it in one single connection rather than divide it among multiple connections. The web http protocol violates this routinely, though. With multiple short connections, `cwnd` may never properly converge to the steady state for any of them; TCP Reno does not support carrying over what has been learned about `cwnd` from one connection to the next. A related issue occurs when a connection alternates between relatively idle periods and full-on data transfer; most TCPs set `cwnd`=1 and return to slow start when sending resumes after an idle period.

Finally, TCP's Fast Retransmit assumes that routers do not significantly reorder packets.

19.10 TCP Parameters

In TCP Reno's Additive Increase, Multiplicative Decrease strategy, the increase increment is 1.0 and the decrease factor is 1/2. It is natural to ask if these values have some especial significance, or what are the consequences if they are changed.

Neither of these values plays much of a role in determining the average value of `cwnd`, at least in the short term; this is largely dictated by the path capacity, including the queue size of the bottleneck router. It seems clear that the exact value of the increase increment has no bearing on congestion; the per-RTT increase is too small to have a major effect here. The decrease factor of 1/2 *may* play a role in responding promptly to incipient congestion, in that it reduces `cwnd` sharply at the first sign of lost packets. However, as we shall see in *22.6 TCP Vegas*, TCP Vegas in its "normal" mode manages quite successfully with an Additive Decrease strategy, decrementing `cwnd` by 1 at the point it detects approaching congestion (to be sure, it detects this well before packet loss), and, by some measures, responds better to congestion than TCP Reno. In other words, not only is the exact value of the AIMD decrease factor not critical for congestion management, but multiplicative decrease itself is not mandatory.

There are two informal justifications in [JK88] for a decrease factor of 1/2. The first is in slow start: if at the Nth RTT it is found that `cwnd` = 2^N is too big, the sender falls back to `cwnd`/2 = 2^{N-1}, which is known to have worked without losses the previous RTT. However, a change here in the decrease policy might best be addressed with a concomitant change to slow start; alternatively, the reduction factor of 1/2 might be left still to apply to "unbounded" slow start, while a new factor of β might apply to threshold slow start.

The second justification for the reduction factor of 1/2 applies directly to the congestion avoidance phase; written in 1988, it is quite remarkable to the modern reader:

> If the connection is steady-state running and a packet is dropped, it's probably because a new connection started up and took some of your bandwidth.... [I]t's probable that there are now exactly two conversations sharing the bandwidth. I.e., you should reduce your window by half because the bandwidth available to you has been reduced by half. [JK88], §D

Today, busy routers may have thousands of simultaneous connections. To be sure, Jacobson and Karels go on to state, "if there are more than two connections sharing the bandwidth, halving your window is conservative – and being conservative at high traffic intensities is probably wise". This advice remains apt today.

But while they do not play a large role in setting `cwnd` or in avoiding "congestive collapse", it turns out that these increase-increment and decrease-factor values of 1 and 1/2 respectively play a *great* role in **fairness**: making sure competing connections get the bandwidth allocation they "should" get. We will return to this in *20.3 TCP Reno Fairness with Synchronized Losses*, and also *21.4 AIMD Revisited*.

19.11 Epilog

TCP Reno's core congestion algorithm is based on algorithms in Jacobson and Karel's 1988 paper [JK88], now twenty-five years old, although NewReno and SACK have been almost universally added to the standard "Reno" implementation.

There are also broad changes in TCP usage patterns. Twenty years ago, the vast majority of all TCP traffic represented downloads from "major" servers. Today, over half of all Internet TCP traffic is peer-to-peer

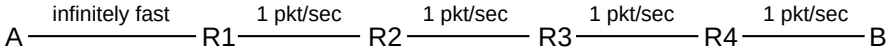
rather than server-to-client. The rise in online video streaming creates new demands for excellent TCP real-time performance.

In the next chapter we will examine the dynamic behavior of TCP Reno, focusing in particular on fairness between competing connections, and on other problems faced by TCP Reno senders. Then, in *22 Newer TCP Implementations*, we will survey some attempts to address these problems.

19.12 Exercises

Exercises may be given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercises marked with a ♢ have solutions or hints at *34.14 Solutions for TCP Reno and Congestion Management*.

1.0. Consider the following network, with each link other than the first having a bandwidth delay of 1 packet/second. Assume ACKs travel instantly from B to R (and thus to A). Assume there are no propagation delays, so the RTT_{noLoad} is 4; the bandwidth×RTT product is then 4 packets. If A uses sliding windows with a window size of 6, the queue at R1 will eventually have size 2.



Suppose A uses **threshold** slow start (*19.2.2 Threshold Slow Start*) with `ssthresh` = 6, and with `cwnd` initially 1. Complete the table below until two rows after `cwnd` = 6; for these final two rows, `cwnd` has reached `ssthresh` and so A will send only one new packet for each ACK received. Assume the queue at R1 is large enough that no packets are dropped. How big will the queue at R1 grow?

T	A sends	R1 queues	R1 sends	B receives/ACKs	`cwnd`
0	1		1		1
1					
2					
3					
4	2,3	3	2	1	2
5			3		2
6					
7					
8	4,5	5	4	2	3

Note that if, instead of using slow start, A simply sends the initial windowful of 6 packets all at once, then the queue at R1 will initially hold 6-1 = 5 packets.

2.0. Consider the following network from *19.2.3 Slow-Start Multiple Drop Example*, with links labeled with bandwidths in packets/ms. Assume ACKs travel instantly from B to R (and thus to A).

A begins sending to B using unbounded slow start, beginning with Data[1] at T=0. Initially, `cwnd` = 1. Write out a table of packet transmissions and deliveries assuming R's maximum queue size is 4 (not counting the packet currently being forwarded). Stop with the arrival at A of the first dupACK triggered by the arrival at B of the packet that followed the first packet that was dropped by R. No retransmissions will occur by then.

T	A sends	R queues	R drops	R sends	B receives/ACKs
0	Data[1]			Data[1]	

3.0. Consider the network from exercise 2.0 above. A again begins sending to B using unbounded slow start, but this time R's queue size is 2, not counting the packet currently being forwarded. Make a table showing all packet transmissions by A, all packet drops by R, and other columns as are useful. Assume no **re**transmission mechanism is used at all (no timeouts, no fast retransmit), and that A sends new data only when it receives new ACKs (dupACKs, in other words, do not trigger new data transmissions). With these assumptions, new data transmissions will eventually cease; continue the table until all transmitted data packets are received by B.

4.0. Suppose a connection starts with `cwnd`=1 and increments `cwnd` by 1 each RTT with no loss, and sets `cwnd` to `cwnd`/2, rounding down, on each RTT with at least one loss. Lost packets are **not retransmitted**, and propagation delays dominate so each windowful is sent more or less together. Packets 5, 13, 14, 23 and 30 are lost. What is the window size each RTT, up until the first 40 packets are sent? What packets are sent each RTT? Hint: in the first RTT, Data[1] is sent. There is no loss, so in the second RTT `cwnd` = 2 and Data[2] and Data[3] are sent.

5.0. Suppose TCP Reno is used to transfer a large file over a path with bandwidth high enough that, during slow start, `cwnd` can be treated as doubling each RTT as in *19.2 Slow Start*. Assume the receiver places no limits on window size.

(a). How many RTTs will it take for the window size to first reach ~8,000 packets (about 2^{13}), assuming unbounded slow start is used and there are no packet losses?

(b). Approximately how many packets will have been sent and acknowledged by that point?

(c). Now assume the bandwidth is 100 packets/ms and the RTT is 80 ms, making the bandwidth×delay product 8,000 packets. What fraction of the total bandwidth will have been used by the connection up to the point where the window size reaches 8000? Hint: the total bandwidth is 8,000 packets per RTT.

6.0. (a) Repeat the diagram in *19.4 TCP Reno and Fast Recovery*, done there with `cwnd`=10, for a window size of 8. Assume as before that the lost packet is Data[10]. There will be seven dupACK[9]'s, which it may be convenient to tag as dupACK[9]/11 through dupACK[9]/17. Be sure to indicate clearly when sending resumes.

(b). Suppose you try to do this with a window size of 6. Is this window size big enough for Fast Recovery still to work? If so, at what dupACK[9]/N does new data transmission begin? If not, what goes wrong?

7.0. Suppose the window size is 100, and Data[1001] is lost. There will be 99 dupACK[1000]'s sent, which we may denote as dupACK[1000]/1002 through dupACK[1000]/1100. TCP Reno is used.

(a). At which dupACK[1000]/N does the sender start sending new data?

(b). When the retransmitted data[1001] arrives at the receiver, what ACK is sent in response?

(c). When the acknowledgment in (b) arrives back at the sender, what data packet is sent?

Hint: express EFS in terms of dupACK[1000]/N, for N⩾1004. The third dupACK is dupACK[1000]/1004; what is EFS at that point after retransmission of Data[1001]?

8.0. Suppose the window size is 40, and Data[1001] is lost. Packet 1000 will be ACKed normally. Packets 1001-1040 will be sent, and 1002-1040 will each trigger a duplicate ACK[1000].

(a). What actual data packets trigger the first three dupACKs? (The first ACK[1000] is triggered by Data[1000]; don't count this one as a duplicate.)

(b). After the third dupACK[1000] has been received and the lost data[1001] has been retransmitted, how many packets/ACKs should the sender estimate as in flight?

When the retransmitted Data[1001] arrives at the receiver, ACK[1040] will be sent back.

(c). What is the first Data[N] sent for which the response is ACK[N], for N>1000?

(d). What is the first N for which Data[N+20] is sent in response to ACK[N] (this represents the point when the connection is back to normal sliding windows, with a window size of 20)?

9.0. Recall (*19.2 Slow Start*) that during slow start `cwnd` is incremented by 1 for each arriving ACK, resulting in the transmission of two new data packets. Suppose slow-start is modified so that, on each ACK, *three* new packets are sent rather than two; `cwnd` will now triple after each RTT, taking values 1, 3, 9, 27, ...

(a). For each arriving ACK, by how much must `cwnd` now be incremented?

(b). Suppose a path has mostly propagation delay. Progressively larger windowfuls are sent, with sizes successive powers of 3, until a `cwnd` is reached where a packet loss occurs. What window size can the sender be reasonably sure *does* work, based on earlier experience?

10.0. Suppose in the example of *19.5 TCP NewReno*, Data[4] had *not* been lost.

(a). When Data[1] is received, what ACK would be sent in response?

(b). At what point in the diagram is the sender able to resume ordinary sliding windows with `cwnd` = 6?

11.0. Suppose in the example of *19.5 TCP NewReno*, Data[1] and Data[2] had been lost, but not Data[4].

(a). The third dupACK[0] is sent in response to what Data[N]?
(b). When the retransmitted Data[1] reaches the receiver, ACK[1] is the response. When this ACK[1] reaches the sender, which Data packets are sent in response?

12.0. Suppose two TCP connections have the same RTT and share a bottleneck link, on which there is no other traffic. The size of the bottleneck queue is negligible when compared to the bandwidth × $\text{RTT}_{\text{noLoad}}$ product. Loss events occur at regular intervals, and are completely synchronized. Show that the two connections together will use 75% of the total bottleneck-link capacity, as in *19.7 TCP and Bottleneck Link Utilization* (there done for a single connection).

See also Exercise 16.0 of chapter *21 Further Dynamics of TCP*.

13.0. In *19.7 TCP and Bottleneck Link Utilization* we showed that, if the bottleneck router queue capacity was 50% of a TCP Reno connection's transit capacity, and there was no other traffic, then the bottleneck-link utilization would be 95.8%.

(a). Suppose the queue capacity is 1/3 of the transit capacity. Show the bottleneck link utilization is 11/12, or 91.7%. Draw a diagram of the tooth, and find the relative lengths of the link-unsaturated and queue-filling phases. You may round off $\texttt{cwnd}_{\text{max}}$ to 4/3 the transit capacity (the value of `cwnd` just before the packet loss; the exact value of $\texttt{cwnd}_{\text{max}}$ is higher by 1).
(b).♢ Derive a formula for the link utilization in terms of the ratio **f**<1 of queue capacity to transit capacity. Make the same simplifying assumption as in part (a).

14.0. In *19.2.1 TCP Reno Per-ACK Responses* we stated that the per-ACK response of a TCP sender was to increment `cwnd` as follows:

```
cwnd = cwnd + 1/cwnd
```

(a). What is the corresponding formulation if the window size is in fact measured in bytes rather than packets? Let `SMSS` denote the sender's maximum segment size, and let `bwnd` = `SMSS`×`cwnd` denote the congestion window as measured in bytes. Hint: solve this last equation for `cwnd` and plug the result in above.
(b). What is the appropriate formulation of `cwnd` = `cwnd` + 1/`cwnd` if delayed ACKs are used (*18.8 TCP Delayed ACKs*) and we still want `cwnd` to be incremented by 1 for each windowful? Assume we are back to measuring `cwnd` in packets.

15.0. In *19.8 Single Packet Losses* we simplified the argument slightly by assuming that when A sent a pair of packets, they arrived at R "essentially simultaneously".

Give a scenario in which it is not the "extra" packet (the second of the pair) that is lost, but the packet that follows it. Hint: see *31.3.4.1 Single-sender phase effects*.

20 DYNAMICS OF TCP

In this chapter we introduce, first and foremost, the possibility that there are other TCP connections out there competing with us for throughput. In *8.3 Linear Bottlenecks* (and in *19.7 TCP and Bottleneck Link Utilization*) we looked at the performance of TCP through an *uncontested* bottleneck; now we allow for competition.

Although the focus of this chapter is on TCP Reno, many, though not all, of the ideas presented here apply to non-Reno TCPs as well, and even to non-TCP mechanisms such as QUIC. Several non-Reno TCP alternatives are presented later in *22 Newer TCP Implementations*.

The following chapter continues this thread, with some more examples of TCP-Reno large-scale behavior.

20.1 A First Look At Queuing

In what order do we transmit the packets in a router's outbound-interface queue? The conventional answer is in the order of arrival; technically, this is **FIFO** (First-In, First-Out) queuing. What happens to a packet that arrives at a router whose queue for the desired outbound interface is full? The conventional answer is that it is dropped; technically, this is known as **tail-drop**.

While **FIFO tail-drop** remains very important, there are alternatives. In an admittedly entirely different context (the IPv6 equivalent of ARP), **RFC 4681** states, "When a queue overflows, the new arrival SHOULD replace the oldest entry." This might be called "head drop"; it is not used for *router* queues.

Queuing Theory

While there are lots of queues in this chapter, we avoid the language of traditional queuing theory. That's because queues created through sliding windows are highly deterministic in terms of both arrivals and departures (so-called "D/D/1" queues). From a queuing-theory perspective, interesting queues tend to have random (*eg* Poisson) arrival or service times, or both; *eg* "M/M/1" queues. The advantage, for us, of sliding-windows queues is that they are a good deal more tractable than the general case. When analyzing independent *messages*, however, M/M/1 queuing is much more important; see for example [LK78].

An alternative drop-policy mechanism that *has* been considered for router queues is **random drop**. Under this policy, if a packet arrives but the destination queue is full, with N packets waiting, then one of the N+1 packets in all – the N waiting plus the new arrival – is chosen at random for dropping. The most recent arrival has thus a very good chance of gaining an initial place in the queue, but also a reasonable chance of being dropped later on. See [AM90].

While random drop is seldom if ever put to production use its original form, it does resolve a peculiar synchronization problem related to TCP's natural periodicity that can lead to starvation for one connection. This situation – known as **phase effects** – will be revisited in *31.3.4 Phase Effects*. Mathematically, random-drop queuing is sometimes more tractable than tail-drop because a packet's loss probability has little dependence on arrival-time race conditions with other packets.

20.1.1 Priority Queuing

A quite different alternative to FIFO is **priority queuing**. We will consider this in more detail in *23.3 Priority Queuing*, but the basic idea is straightforward: whenever the router is ready to send the next packet, it looks first to see if it has any higher-priority packets to send; lower-priority packets are sent only when there is no waiting higher-priority traffic. This can, of course, lead to complete starvation for the lower-priority traffic, but often there are bandwidth constraints on the higher-priority traffic (*eg* that it amounts to less than 10% of the total available bandwidth) such that starvation does not occur.

In an environment of mixed real-time and bulk traffic, it is natural to use priority queuing to give the real-time traffic priority service, by assignment of such traffic to the higher-priority queue. This works quite well as long as, say, the real-time traffic is less than some fixed fraction of the total; we will return to this in *25 Quality of Service*.

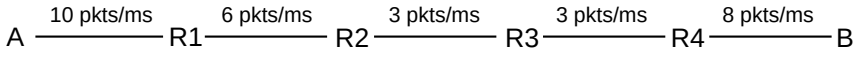
20.2 Bottleneck Links with Competition

So far we have been ignoring the fact that there are other TCP connections out there. A single connection in isolation needs not to overrun its bottleneck router and drop packets, at least not too often. However, once there are other connections present, then each individual TCP connection also needs to consider how to maximize its share of the aggregate bandwidth.

Consider a simple network path, with bandwidths shown in packets/ms. The minimum bandwidth, or **path bandwidth**, is 3 packets/ms.

20.2.1 Example 1: linear bottleneck

Below is the example we considered in *8.3 Linear Bottlenecks*; bandwidths are shown in packets/ms.

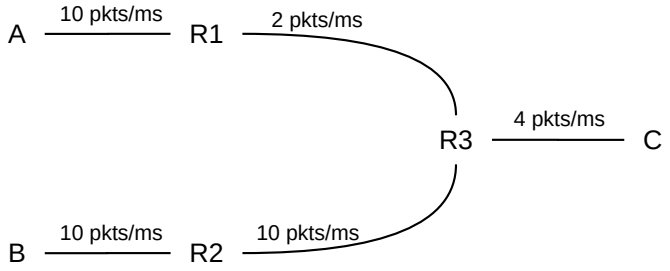


The bottleneck link for A→B traffic is at R2, and the queue will form at R2's outbound interface.

We claimed earlier that if the sender uses sliding windows with a fixed window size, then the network will converge to a steady state in relatively short order. This is also true if multiple senders are involved; however, a mathematical proof of convergence may be more difficult.

20.2.2 Example 2: router competition

The bottleneck-link concept is a useful one for understanding congestion due to a single connection. However, if there are multiple senders in **competition** for a link, the situation is more complicated. Consider the following diagram, in which links are labeled with bandwidths in packets/ms:



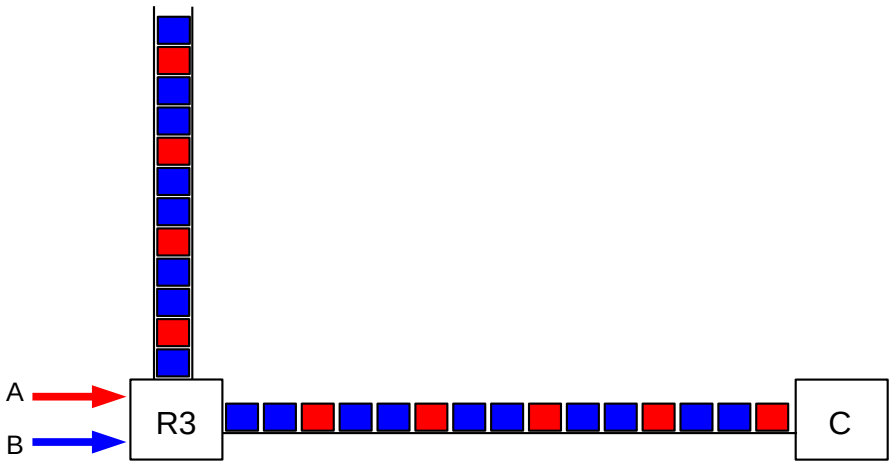
For a moment, assume R3 uses *priority* queuing, with the B⟶C path given priority over A⟶C. If B's flow to C is fixed at 3 packets/ms, then A's share of the R3–C link will be 1 packet/ms, and A's bottleneck will be at R3. However, if B's total flow rate drops to 1 packet/ms, then the R3–C link will have 3 packets/ms available, and the bottleneck for the A–C path will become the 2 packet/ms R1–R3 link.

Now let us switch to the more-realistic *FIFO* queuing at R3. If B's flow is 3 packets/ms and A's is 1 packet/ms, then the R3–C link will be saturated, but just barely: if each connection sticks to these rates, no queue will develop at R3. However, it is no longer accurate to describe the 1 packet/ms as A's *share*: if A wishes to send more, it will begin to compete with B. At first, the queue at R3 will grow; eventually, it is quite possible that B's total flow rate might drop because *B is losing to A in the competition for R3's queue*. This latter effect is very real.

In general, if two connections share a bottleneck link, they are competing for the bandwidth of that link. That bandwidth share, however, is *precisely dictated by the queue share as of a short while before*. R3's fixed rate of 4 packets/ms means one packet every 250 µs. If R3 has a queue of 100 packets, and in that queue there are 37 packets from A and 63 packets from B, then over the next 25 ms (= 100 × 250 µs) R3's traffic to C will consist of those 37 packets from A and the 63 from B. Thus the competition between A and B for R3–C bandwidth is *first fought as a competition for R3's queue space*. This is important enough to state as as rule:

> **Queue-Competition Rule**: in the steady state, if a connection utilizes fraction $\alpha \leqslant 1$ of a FIFO router's queue, then that connection has a share of α of the router's total outbound bandwidth.

Below is a picture of R3's queue and outbound link; the queue contains four packets from A and eight from B. The link, too, contains packets in this same ratio; presumably packets from B are consistently arriving twice as fast as packets from A.



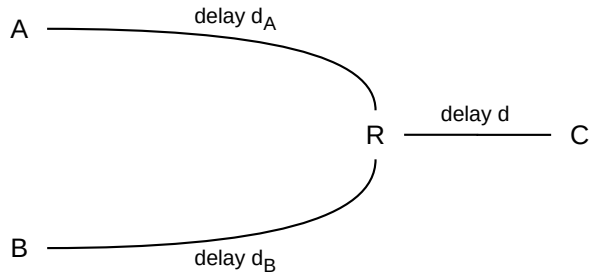
In the steady state here, A and B will use four and eight packets, respectively, of R3's queue capacity. As acknowledgments return, each sender will replenish the queue accordingly. However, it is not in A's long-term interest to settle for a queue utilization at R3 of four packets; A may want to take steps that will lead in this setting to a gradual increase of its queue share.

Although we started the discussion above with fixed packet-sending **rates** for A and B, in general this leads to instability. If A and B's combined rates add up to more than 4 packets/ms, R3's queue will grow without bound. It is much better to have A and B use **sliding windows**, and give them each fixed window sizes; in this case, as we shall see, a stable equilibrium is soon reached. Any combination of window sizes is legal regardless of the available bandwidth; the queue utilization (and, if necessary, the loss rate) will vary as necessary to adapt to the actual bandwidth.

If there are several competing flows, then a given connection may have multiple bottlenecks, in the sense that there are several routers on the path experiencing queue buildups. In the steady state, however, we can still identify the link (or first link) with minimum bandwidth; we can call this link the bottleneck. Note that the bottleneck link in this sense can change with the sender's winsize and with competing traffic.

20.2.3 Example 3: competition and queue utilization

In the next diagram, the bottleneck R–C link has a normalized bandwidth of 1 packet per ms (or, more abstractly, one packet per unit time). The bandwidths of the A–R and B–R links do not matter, except they are greater than 1 packet per ms. Each link is labeled with the **propagation delay**, measured in the same time unit as the bandwidth; the delay thus represents the number of packets the link can be transporting at the same time, if sent at the bottleneck rate.



The network layout here, with the shared R–C link as the bottleneck, is sometimes known as the **singlebell** topology. A perhaps-more-common alternative is the **dumbbell** topology of *20.3 TCP Reno Fairness with Synchronized Losses*, though the two are equivalent for our purposes.

Suppose A and B each send to C using sliding windows, each with **fixed** values of winsize w_A and w_B. Suppose further that these winsize values are large enough to saturate the R–C link. *How big will the queue be at R?* And how will the bandwidth divide between the A⟶C and B⟶C flows?

For the two-competing-connections example above, assume we have reached the steady state. Let α denote the fraction of the bandwidth that the A⟶C connection receives, and let β = 1-α denote the fraction that the B⟶C connection gets; because of our normalization choice for the R–C bandwidth, α and β also represent respective throughputs. From the Queue-Competition Rule above, these bandwidth proportions must agree with the queue proportions; if Q denotes the combined queue utilization of both connections, then that queue will have about αQ packets from the A⟶C flow and about βQ packets from the B⟶C flow.

We worked out the queue usage precisely in *8.3.2 RTT Calculations* for a *single* flow; we derived there the following:

$$\text{queue\_usage} = \text{winsize} - \text{throughput} \times \text{RTT}_{\text{noLoad}}$$

where we have here used "throughput" instead of "bandwidth" to emphasize that this is the dynamic share rather than the physical transmission capacity.

This equation remains true for each separate flow in the present case, where the $\text{RTT}_{\text{noLoad}}$ for the A⟶C connection is $2(d_A+d)$ (the factor of 2 is to account for the round-trip) and the $\text{RTT}_{\text{noLoad}}$ for the B⟶C connection is $2(d_B+d)$. We thus have

$$\alpha Q = w_A - 2\alpha(d_A+d)$$

$$\beta Q = w_B - 2\beta(d_B+d)$$

or, alternatively,

$$\alpha[Q + 2d + 2d_A] = w_A$$

$$\beta[Q + 2d + 2d_B] = w_B$$

If we add the first pair of equations above, we can obtain the combined queue utilization:

$$Q = w_A + w_B - 2d - 2(\alpha d_A + \beta d_B)$$

The last term here, $2(\alpha d_A+\beta d_B)$, represents the number of A's packets in flight on the A–R link plus the number of B's packets in flight on the B–R link.

We can solve these equations exactly for α, β and Q in terms of the known quantities, but the algebraic solution is not particularly illuminating. Instead, we examine a few more-tractable special cases.

20.2.3.1 The equal-delays case

We consider first the special case of **equal delays**: $d_A = d_B = d'$. In this case the term $(\alpha d_A + \beta d_B)$ simplifies to d', and thus we have $Q = w_A + w_B - 2d - 2d'$. Furthermore, if we divide corresponding sides of the second pair of equations above, we get $\alpha/\beta = w_A/w_B$; that is, the bandwidth (and thus the queue utilization) divides in exact accordance to the window-size proportions.

If, however, d_A is larger than d_B, then a greater fraction of the A⟶C packets will be in transit, and so fewer will be in the queue at R, and so α will be somewhat smaller and β somewhat larger.

20.2.3.2 The equal-windows case

If we assume equal winsize values instead, $w_A = w_B = w$, then we get

$$\alpha/\beta = [Q + 2d + 2d_B] / [Q + 2d + 2d_A]$$

The bandwidth ratio here is biased against the larger of d_A or d_B. That is, if $d_A > d_B$, then more of A's packets will be in transit, and thus fewer will be in R's queue, and so A will have a smaller fraction of the the bandwidth. This bias is, however, not quite proportional: if we assume d_A is double d_B and $d_B = d = Q/2$, then $\alpha/\beta = 3/4$, and A gets 3/7 of the bandwidth to B's 4/7.

Still assuming $w_A = w_B = w$, let us decrease w to the point where the link is just saturated, but Q=0. At this point $\alpha/\beta = [d+d_B]/[d+d_A]$; that is, bandwidth divides according to the respective RTT_{noLoad} values. As w rises, additional queue capacity is used and α/β will move closer to 1.

20.2.3.3 The fixed-w_B case

Finally, let us consider what happens if w_B is **fixed** at a large-enough value to create a queue at R from the B–C traffic alone, while w_A then increases from zero to a point much larger than w_B. Denote the number of B's packets in R's queue by Q_B; with $w_A = 0$ we have β=1 and $Q = Q_B = w_B - 2(d_B+d)$ = throughput × $(RTT - RTT_{noLoad})$.

As w_A begins to increase from zero, the competition will decrease B's throughput. We have $\alpha = w_A/[Q+2d+2d_A]$; **small** changes in w_A will not lead to much change in Q, and even less in $Q+2d+2d_A$, and so α will initially be approximately proportional to w_A.

For B's part, increased competition from A (increased w_A) will always decrease B's share of the bottleneck R–C link; this link is saturated and every packet of A's in transit there must take away one slot on that link for a packet of B's. This in turn means that B's bandwidth β must decrease as w_A rises. As B's bandwidth decreases, $Q_B = \beta Q = w_B - 2\beta(d_B+d)$ must increase; another way to put this is as the transit capacity falls, the queue utilization rises. For $Q_B = \beta Q$ to increase while β decreases, Q must be increasing faster than β is decreasing.

Finally, we can conclude that as w_A gets large and $\beta \to 0$, the limiting value for B's queue utilization Q_B at R will be the entire windowful w_B, up from its starting value (when w_A=0) of $w_B - 2(d_B+d)$. If d_B+d had been

small relative to w_B, then Q_B's increase will be modest, and it may be appropriate to consider Q_B relatively constant.

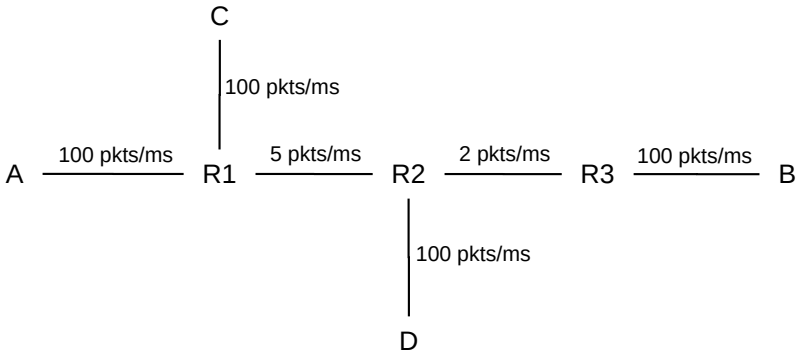
We remark again that the formulas here are based on the assumption that the bottleneck bandwidth is one packet per unit time; see exercise 1.0 for the necessary adjustments for conventional bandwidth measurements.

20.2.3.4 The iterative solution

Given d, d_A, d_B, w_A and w_B, one way to solve for α, β and Q is to proceed **iteratively**. Suppose an initial $\langle\alpha,\beta\rangle$ is given, as the respective fractions of packets in the queue at R. Over the next period of time, α and β must (by the Queue Rule) become the bandwidth ratios. If the A–C connection has bandwidth α (recall that the R–C connection has bandwidth 1.0, in packets per unit time, so a bandwidth fraction of α means an actual bandwidth of α), then the number of packets in bidirectional transit will be $2\alpha(d_A+d)$, and so the number of A–C packets in R's queue will be $Q_A = w_A - 2\alpha(d_A+d)$; similarly for Q_B. At that point we will have $\alpha_{new} = Q_A/(Q_A+Q_B)$. Starting with an appropriate guess for α and iterating $\alpha \rightarrow \alpha_{new}$ a few times, if the sequence converges then it will converge to the steady-state solution. Convergence is not guaranteed, however, and is dependent on the initial guess for α. One guess that often leads to convergence is $w_A/(w_A+w_B)$.

20.2.4 Example 4: cross traffic and RTT variation

In the following diagram, let us consider what happens to the A–B traffic when the C⟶D link ramps up. Bandwidths shown are expressed as packets/ms and all queues are FIFO. (Because the bandwidth is not equal to 1.0, we cannot apply the formulas of the previous section directly.) We will assume that propagation delays are small enough that only an inconsequential number of packets from C to D can be simultaneously in transit at the bottleneck rate of 5 packets/ms. All senders will use sliding windows.



Let us suppose the A–B link is idle, and the C⟶D connection begins sending with a window size chosen so as to create a queue of 30 of C's packets at R1 (if propagation delays are such that two packets can be in transit each direction, we would achieve this with winsize=34).

Now imagine A begins sending. If A sends a single packet, is not shut out even though the R1–R2 link is 100% busy. A's packet will simply have to wait at R1 behind the 30 packets from C; the waiting time in the

queue will be 30 packets ÷ (5 packets/ms) = 6 ms. If we change the winsize of the C⟶D connection, the delay for A's packets will be directly proportional to the number of C's packets in R1's queue.

To most intents and purposes, the C⟶D flow here has increased the RTT of the A⟶B flow by 6 ms. As long as A's contribution to R1's queue is small relative to C's, the delay at R1 for A's packets looks more like propagation delay than bandwidth delay, because if A sends two back-to-back packets they will likely be enqueued consecutively at R1 and thus be subject to a single 6 ms queuing delay. By varying the C⟶D window size, we can, within limits, increase or decrease the RTT for the A⟶B flow.

Let us return to the fixed C⟶D window size – denoted w_C – chosen to yield a queue of 30 of C's packets at R1. As A increases its own window size from, say, 1 to 5, the C⟶D throughput will decrease slightly, but C's contribution to R1's queue will remain dominant.

As in the argument at the end of *20.2.3.3 The fixed-wB case*, small propagation delays mean that w_C will not be much larger than 30. As w_A climbs from zero to infinity, C's contribution to R1's queue rises from 30 to at most w_C, and so the 6ms delay for A⟶B packets remains relatively constant even as A's winsize rises to the point that A's contribution to R1's queue far outweighed C's. (As we will argue in the next paragraphs, this can actually happen only if the R2–R3 bandwidth is increased). Each packet from A arriving at R1 will, on average, face 30 or so of C's packets ahead of it, along with anywhere from many fewer to many more of A's packets.

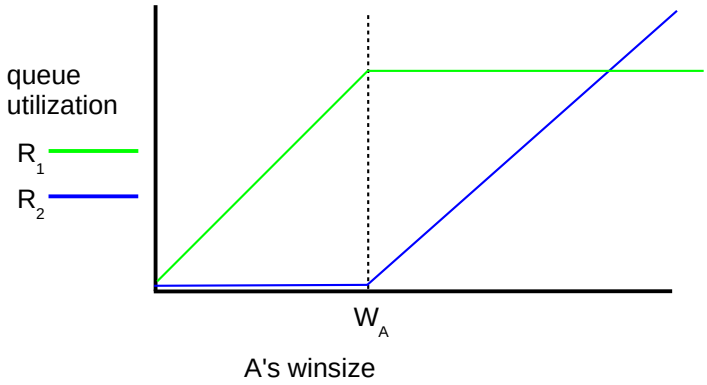
If A's window size is 1, its one packet at a time will wait 6 ms in the queue at R1. If A's window size is greater than 1 but remains small, so that A contributes only a small proportion of R1's queue, then A's packets will wait only at R1. Initially, as A's winsize increases, the queue at R1 grows but all other queues remain empty.

However, if A's winsize grows large enough that its packets consume 40% of R1's queue in the steady state, then this situation changes. At the point when A has 40% of R1's queue, by the Queue Competition Rule it will also have a 40% share of the R1–R2 link's bandwidth, that is, 40% × 5 = 2 packets/ms. Because the R2–R3 link has a bandwidth of 2 packets/ms, *the A–B throughput can never grow beyond this*. If the C–D contribution to R1's queue is held constant at 30 packets, then this point is reached when A's contribution to R1's queue is 20 packets.

Because A's proportional contribution to R1's queue cannot increase further, any additional increase to A's winsize must result in those packets now being enqueued at R2.

We have now reached a situation where A's packets are queuing up at both R1 and at R2, contrary to the single-sender principle that packets can queue at only one router. Note, however, that for any fixed value of A's winsize, a small-enough increase in A's winsize will result in either that increase going entirely to R1's queue or entirely to R2's queue. Specifically, if w_A represents A's winsize at the point when A has 40% of R1's queue (a little above 20 packets if propagation delays are small), then for winsize < w_A any queue growth will be at R1 while for winsize > w_A any queue growth will be at R2. In a sense the bottleneck link "switches" from R1–R2 to R2–R3 at the point winsize = w_A.

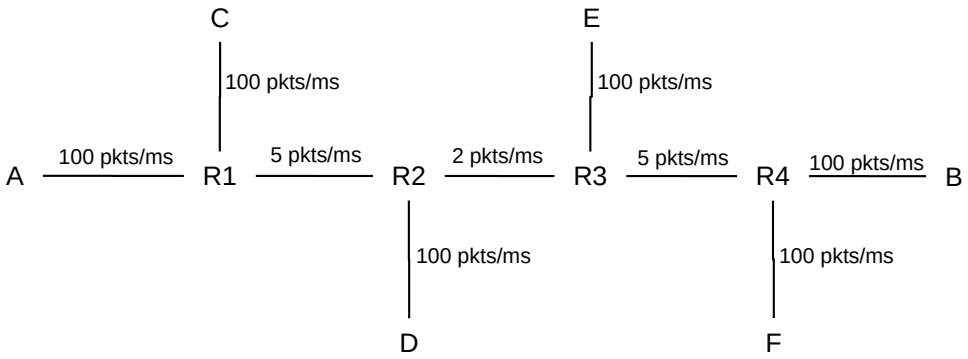
In the graph below, A's contribution to R1's queue is plotted in green and A's contribution to R2's queue is in blue. It may be instructive to compare this graph with the third graph in *8.3.3 Graphs at the Congestion Knee*, which illustrates a single connection with a single bottleneck.



In Exercise 8.0 we consider some minor changes needed if propagation delay is *not* inconsequential.

20.2.5 Example 5: dynamic bottlenecks

The next example has two links offering potential competition to the A⟶B flow: C⟶D and E⟶F. Either of these could send traffic so as to throttle (or at least compete with) the A⟶B traffic. Either of these could choose a window size so as to build up a persistent queue at R1 or R3; a persistent queue of 20 packets would mean that A⟶B traffic would wait 4 ms in the queue.



Despite situations like this, we will usually use the term "bottleneck link" as if it were a precisely defined concept. In Examples 2, 3 and 4 above, a better term might be "competitive link"; for Example 5 we perhaps should say "competitive link**s**.

20.2.6 Packet Pairs

One approach for a sender to attempt to measure the physical bandwidth of the bottleneck link is the **packet-pairs** technique: the sender repeatedly sends a pair of packets P1 and P2 to the receiver, one right after the other. The receiver records the time difference between the arrivals.

Sooner or later, we would expect that P1 and P2 would arrive consecutively at the bottleneck router R, and be put into the queue next to each other. They would then be sent one right after the other on the bottleneck

link; if T is the time difference in arrival at the far end of the link, the physical bandwidth is size(P1)/T. At least some of the time, the packets will remain spaced by time T for the rest of their journey.

The theory is that the receiver can measure the different arrival-time differences for the different packet pairs, and look for the *minimum* time difference. Often, this will be the time difference introduced by the bandwidth delay on the bottleneck link, as in the previous paragraph, and so the ultimate receiver will be able to infer that the bottleneck physical bandwidth is size(P1)/T.

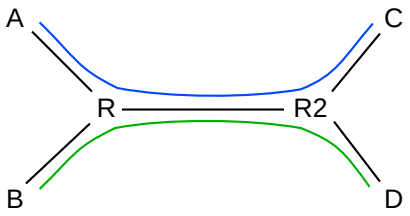
Two things can mar this analysis. First, packets may be reordered; P2 might arrive before P1. Second, P1 and P2 can arrive together at the bottleneck router and be sent consecutively, but then, later in the network, the two packets can arrive at a second router R2 with a (transient) queue large enough that P2 arrives while P1 is in R2's queue. If P1 and P2 are consecutive in R2's queue, then the ultimate arrival-time difference is likely to reflect R2's (higher) outbound bandwidth rather than R's.

Additional analysis of the problems with the packet-pair technique can be found in [VP97], along with a proposal for an improved technique known as *packet bunch mode*.

20.3 TCP Reno Fairness with Synchronized Losses

This brings us to the question of just what *is* a "fair" division of bandwidth. A starting place is to assume that "fair" means "equal", though, as we shall see below, the question does not end there.

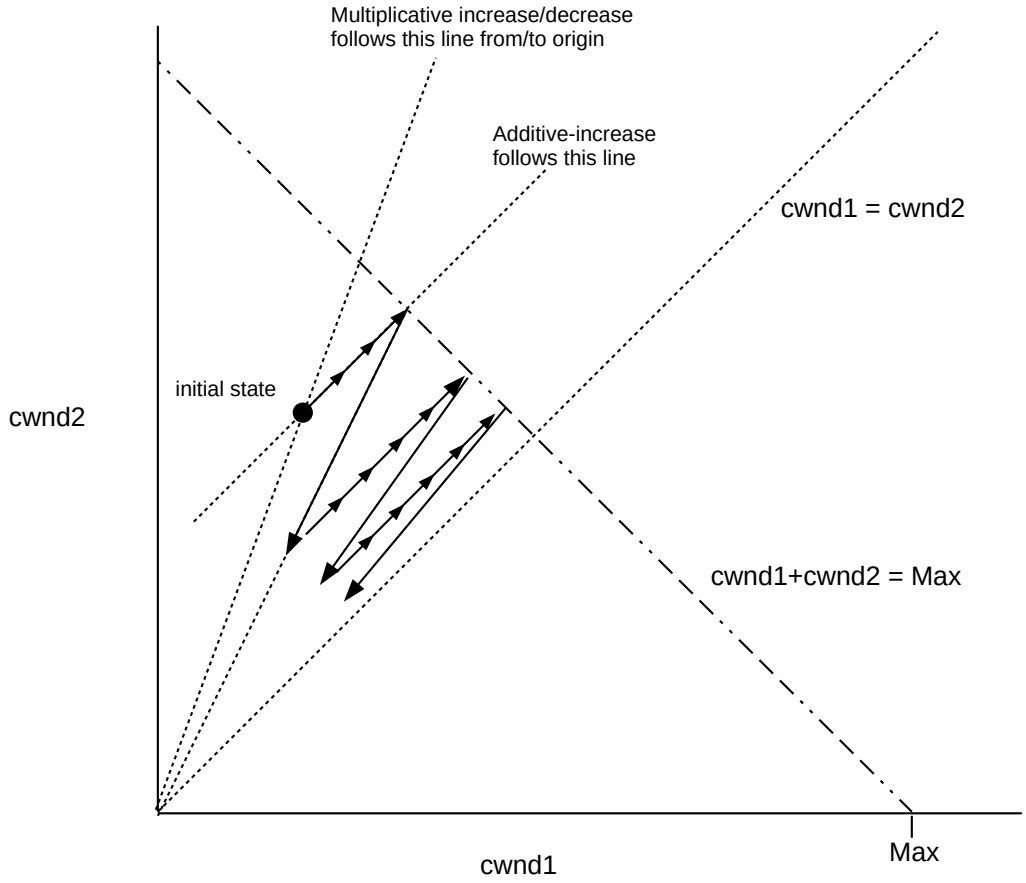
For the moment, consider again two competing TCP Reno connections: Connection 1 (in blue) from A to C and Connection 2 (in green) from B to D, through the same bottleneck router R, *and with the same RTT*. The router R will use tail-drop queuing.



The layout illustrated here, with the shared link somewhere in the middle of each path, is sometimes known as the **dumbbell** topology.

For the time being, we will also continue to assume the TCP Reno **synchronized-loss hypothesis**: that in any one RTT either *both* connections experience a loss or *neither* does. (This assumption is suspect; we explore it further in *20.3.3 TCP Reno RTT bias* and in *31.3 Two TCP Senders Competing*). This was the model reviewed previously in *19.1.1.1 A first look at fairness*; we argued there that in any RTT without a loss, the expression ($\texttt{cwnd}_1$ - $\texttt{cwnd}_2$) remained the same (both `cwnd`s incremented by 1), while in any RTT *with* a loss the expression ($\texttt{cwnd}_1$ - $\texttt{cwnd}_2$) decreased by a factor of 2 (both `cwnd`s decreased by factors of 2).

Here is a graphical version of the same argument, as originally introduced in [CJ89]. We plot $\texttt{cwnd}_1$ on the x-axis and $\texttt{cwnd}_2$ on the y-axis. An additive increase of both (in equal amounts) moves the point (x,y) = ($\texttt{cwnd}_1$,$\texttt{cwnd}_2$) along the line parallel to the 45° line y=x; equal multiplicative decreases of both moves the point (x,y) along a line straight back towards the origin. If the maximum network capacity is Max, then a loss occurs whenever x+y exceeds Max, that is, the point (x,y) crosses the line x+y=Max.



Beginning at the initial state, additive increase moves the state at a 45° angle up to the line x+y=Max, in small increments denoted by the small arrowheads. At this point a loss would occur, and the state jumps back halfway *towards the origin*. The state then moves at 45° incrementally back to the line x+y=Max, and continues to zigzag slowly towards the equal-shares line y=x.

Any attempt to increase `cwnd` faster than linear will mean that the increase phase is not parallel to the line y=x, but in fact veers away from it. This will slow down the process of convergence to equal shares.

Finally, here is a **timeline** version of the argument. We will assume that the A–C path capacity, the B–D path capacity and R's queue size all add up to 24 packets, and that in any RTT in which $\texttt{cwnd}_1 + \texttt{cwnd}_2 > 24$, both connections experience a packet loss. We also assume that, initially, the first connection has `cwnd`=20, and the second has `cwnd`=1.

T	A–C	B–D	
0	20	1	
1	21	2	
2	22	3	total `cwnd` is 25; packet loss
3	11	1	
4	12	2	
5	13	3	

Continued on next page

Table 1 – continued from previous page

T	A–C	B–D	
6	14	4	
7	15	5	
8	16	6	
9	17	7	
10	18	8	second packet loss
11	9	4	
12	10	5	
13	11	6	
14	12	7	
15	13	8	
16	14	9	
17	15	10	third packet loss
18	7	5	
19	8	6	
20	9	7	
21	10	8	
22	11	9	
23	12	10	
24	13	11	
25	14	12	fourth loss
26	7	6	`cwnd`s are quite close
...			
32	13	12	loss
33	6	6	`cwnd`s are equal

So far, fairness seems to be winning.

20.3.1 Example 2: Faster additive increase

Here is the same kind of timeline – again with the synchronized-loss hypothesis – but with the additive-increase increment changed from 1 to 2 for the B–D connection (but not for A–C); both connections start with `cwnd`=1. Again, we assume a loss occurs when $\texttt{cwnd}_1 + \texttt{cwnd}_2 > 24$

T	A–C	B–D	
0	1	1	
1	2	3	
2	3	5	
3	4	7	
4	5	9	
5	6	11	
6	7	13	
7	8	15	
8	9	17	first packet loss
9	4	8	
10	5	10	
11	6	12	
12	7	14	
13	8	16	
14	9	18	second loss
15	4	9	essentially where we were at T=9

The effect here is that the second connection's average `cwnd`, and thus its throughput, is double that of the first connection. Thus, changes to the additive-increase increment lead to very significant changes in fairness. In general, an additive-increase value of α increases throughput, relative to TCP Reno, by a factor of α.

20.3.2 Example 3: Longer RTT

For the next example, we will return to standard TCP Reno, with an increase increment of 1. But here we assume that the RTT of the A–C connection is **double** that of the B–D connection, perhaps because of additional delay in the A–R link. The longer RTT means that the first connection sends packet flights only when T is even. Here is the timeline, where we allow the first connection a hefty head-start. As before, we assume a loss occurs when $\texttt{cwnd}_1 + \texttt{cwnd}_2 > 24$.

T	A–C	B–D	
0	20	1	
1		2	
2	21	3	
3		4	
4	22	5	first loss
5		2	
6	11	3	
7		4	
8	12	5	
9		6	
10	13	7	
11		8	

Continued on next page

Table 2 – continued from previous page

T	A–C	B–D	
12	14	9	
13		10	
14	15	11	second loss
15		5	
16	7	6	
17		7	
18	8	8	B–D has caught up
20	9	10	**from here on only even values for T shown**
22	10	12	
24	11	14	third loss
26	5	8	B–D is now ahead
28	6	10	
30	7	12	
32	8	14	
34	9	16	fourth loss
35		8	
36	4	9	
38	5	11	
40	6	13	
42	7	15	
44	8	17	fifth loss
45		8	
46	4	9	exactly where we were at T=36

The interval $36 \leqslant T < 46$ represents the steady state here; the first connection's average `cwnd` is 6 while the second connection's average is (8+9+...+16+17)/10 = 12.5. Worse, the first connection sends a windowful only half as often. In the interval $36 \leqslant T < 46$ the first connection sends 4+5+6+7+8 = 30 packets; the second connection sends 125. The cost of the first connection's longer RTT is *quadratic*; in general, as we argue more formally below, if the first connection has RTT = $\lambda > 1$ relative to the second's, then its bandwidth will be reduced by a factor of $1/\lambda^2$.

Is this fair?

Early thinking was that there was something to fix here; see [F91] and [FJ92], §3.3 where the Constant-Rate window-increase algorithm is discussed. A more recent attempt to address this problem is **TCP Hybla**, [CF04]; discussed later in *22.12 TCP Hybla*.

Alternatively, we may simply *define* TCP Reno's bandwidth allocation as "fair", at least in some contexts. This approach is particularly common when the issue at hand is making sure other TCP implementations – and non-TCP flows – compete for bandwidth in roughly the same way that TCP Reno does. While TCP Reno's strategy is now understood to be "greedy" in some respects, "fixing" it in the Internet at large is generally recognized as a very difficult option.

20.3.3 TCP Reno RTT bias

Let us consider more carefully the way TCP allocates bandwidth between two connections sharing a bottleneck link with relative RTTs of 1 and λ>1. We claimed above that the slower connection's bandwidth will be reduced by a factor of $1/\lambda^2$; we will now show this under some assumptions. First, uncontroversially, we will assume FIFO droptail queuing at the bottleneck router, and also that the network ceiling (and hence `cwnd` at the point of loss) is "sufficiently" large. We will also assume, for simplicity, that the network ceiling C is constant.

We need one more assumption: that most loss events are experienced by both connections. This is the **synchronized losses** hypothesis, and is the most debatable; we will explore it further in the next section. But first, here is the general argument with this assumption.

Let connection 1 be the faster connection, and assume a steady state has been reached. Both connections experience loss when $\texttt{cwnd}_1+\texttt{cwnd}_2 \geqslant C$, because of the synchronized-loss hypothesis. Let c_1 and c_2 denote the respective window sizes at the point just before the loss. Both `cwnd` values are then halved. Let N be the number of RTTs *for connection 1* before the network ceiling is reached again. During this time c_1 increases by N; c_2 increases by approximately N/λ if N is reasonably large. Each of these increases represents half the corresponding `cwnd`; we thus have $c_1/2 = N$ and $c_2/2 = N/\lambda$. Taking ratios of respective sides, we get $c_1/c_2 = N/(N/\lambda) = \lambda$, and from that we can solve to get $c_1 = C\lambda/(1+\lambda)$ and $c_2 = C/(1+\lambda)$.

To get the relative bandwidths, we have to count packets sent during the interval between losses. Both connections have `cwnd` averaging about 3/4 of the maximum value; that is, the average `cwnd`s are 3/4 c_1 and 3/4 c_2 respectively. Connection 1 has N RTTs and so sends about 3/4 $c_1 \times N$ packets. Connection 2, with its slower RTT, has only about N/λ RTTs (again we use the assumption that N is reasonably large), and so sends about 3/4 $c_2 \times N/\lambda$ packets. The ratio of these is $c_1/(c_2/\lambda) = \lambda^2$. Connection 1 sends fraction $\lambda^2/(1+\lambda^2)$ of the packets; connection 2 sends fraction $1/(1+\lambda^2)$.

20.3.4 Synchronized-Loss Hypothesis

The synchronized-loss hypothesis says that if two TCP connections share a bottleneck link, then whenever the queue is full, late-arriving packets from *each* connection will find it so, and be dropped. Once the queue becomes full, in other words, it stays full for long enough for each connection to experience a packet loss. The hypothesis is most relevent when, as is the case for TCP Reno, packet losses trigger changes to `cwnd`.

This hypothesis is mostly a convenience for reasoning about TCP, and should not be taken as literal fact, though it is often "largely" true. That said, it is certainly possible to come up with hypothetical situations where losses are not synchronized. Recall that a TCP Reno connection's `cwnd` is incremented by only 1 each RTT; losses generally occur when this single extra packet generated by the increment to `cwnd` arrives to find a full queue. Generally speaking, packets are leaving the queue about as fast as they are arriving; actual overfull-queue instants may be rare. It is certainly conceivable that, at least some of the time, one connection would overflow the queue by one packet, and halve its `cwnd`, in a short enough time interval that the other connection misses the queue-full moment entirely. Alternatively, if queue overflows lead to effectively random selection of lost packets (as would certainly be true for random-drop queuing, and might be true for tail-drop if there were sufficient randomness in packet arrival times), then there is a finite probability that all the lost packets at a given loss event come from the same connection.

The synchronized-loss hypothesis is still valid if either or both connection experiences *more* than one packet loss, within a single RTT; the hypothesis fails only when one connection experiences no losses.

We will return to possible failure of the synchronized-loss hypothesis in *21.2.2 Unsynchronized TCP Losses*. In *31.3 Two TCP Senders Competing* we will consider some TCP Reno simulations in which actual measurement does not entirely agree with the synchronized-loss model. Two problems will emerge. The first is that when two connections compete in isolation, a form of synchronization known as **phase effects** (*31.3.4 Phase Effects*) can introduce a persistent perhaps-unexpected bias. The second is that the longer-RTT connection often does manage to miss out on the full-queue moment entirely, as discussed above in the second paragraph of this section. This results in a larger `cwnd` than the synchronized-loss hypothesis would predict.

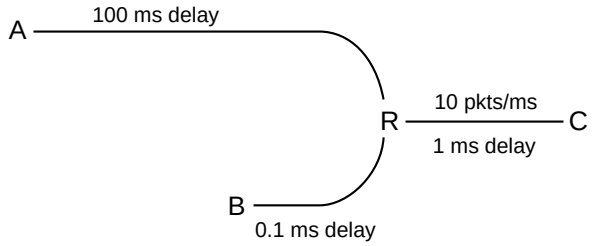
20.3.5 Loss Synchronization

The synchronized-loss hypothesis assumes *all* losses are synchronized. There is another side to this phenomenon that is an issue even if only some reasonable fraction of loss events are synchronized: synchronized losses may represent a collective inefficiency in the use of bandwidth. In the immediate aftermath of a synchronized loss, it is very likely that the bottleneck link will go underutilized, as (at least) two connections using it have just cut their sending rate in half. Better utilization would be achieved if the loss events could be staggered, so that at the point when connection 1 experiences a loss, connection 2 is only halfway to its next loss. For an example, see exercise 18.0 in the following chapter.

This loss synchronization is a very real effect on the Internet, even if losses are not necessarily *all* synchronized. A major contributing factor to synchronization is the relatively slow response of all parties involved to packet loss. In the diagram above at *20.3 TCP Reno Fairness with Synchronized Losses*, if A increments its `cwnd` leading to an overflow at R, the A–R link is likely still full of packets, and R's queue remains full, and so there is a reasonable likelihood that sender B will also experience a loss, even if its `cwnd` was not particularly high, simply because its packets arrived at the wrong instant. Congestion, unfortunately, takes time to clear.

20.3.6 Extreme RTT Ratios

What happens to TCP Reno fairness if one TCP connection has a 100-fold-larger RTT than another? The short answer is that the shorter connection *may* get 10,000 times the throughput. The longer answer is that this isn't quite as easy to set up as one might imagine. For the arguments above, it is necessary for the two connections to have a common bottleneck link:



In the diagram above, the A–C connection wants its `cwnd` to be about 200 ms × 10 packets/ms = 2,000 packets; it is competing for the R–C link with the B–D connection which is happy with a `cwnd` of 22. If R's queue capacity is also about 20, then with most of the bandwidth the B–C connection will experience a loss about every 20 RTTs, which is to say every 22 ms. If the A–C link shares even a modest fraction of those losses, it is indeed in trouble.

However, the A–C `cwnd` cannot fall below 1.0; to test the 10,000-fold hypothesis taking this constraint into account we would have to scale up the numbers on the B–C link so the transit capacity there was at least 10,000. This would mean a 400 Gbps R–C bandwidth, or else an unrealistically large A–R delay.

As a second issue, realistically the A–C link is much more likely to have its bottleneck somewhere in the middle of its long path. In a typical real scenario along the lines of that diagrammed above, B, C and R are all local to a site, and bandwidth of long-haul paths is almost always less than the local LAN bandwidth within a site. If the A–R path has a 1 packet/ms bottleneck somewhere, then it may be less likely to be as dramatically affected by B–C traffic.

A few actual simulations using the methods of *31.3 Two TCP Senders Competing* resulted in an average `cwnd` for the A–C connection of between 1 and 2, versus a B–C `cwnd` of 20-25, regardless of whether the two links shared a bottleneck or if the A–C link had its bottleneck somewhere along the A–R path. This *may* suggest that the A–C connection was indeed saved by the 1.0 `cwnd` minimum.

20.4 Epilog

TCP Reno's core congestion algorithm is based on algorithms in Jacobson and Karel's 1988 paper [JK88], now twenty-five years old. There are concerns both that TCP Reno uses too much bandwidth (the greediness issue) and that it does not use enough (the high-bandwidth-TCP problem).

In the next chapter we consider alternative versions of TCP that attempt to solve some of the above problems associated with TCP Reno.

20.5 Exercises

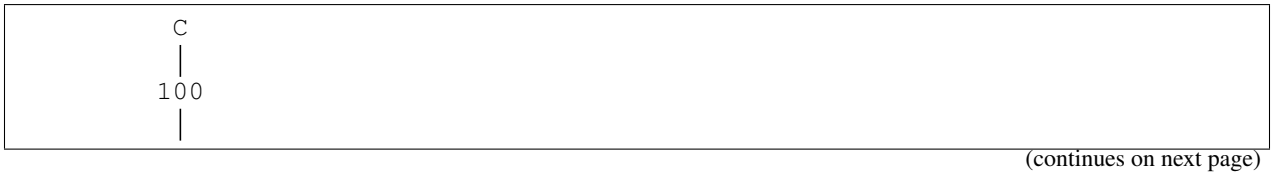
Exercises may be given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercises marked with a ♢ have solutions or hints at *34.15 Solutions for Dynamics of TCP*.

1.0. In the section *20.2.3 Example 3: competition and queue utilization*, we derived the formula

$$Q = w_A + w_B - 2d - 2(\alpha d_A + \beta d_B)$$

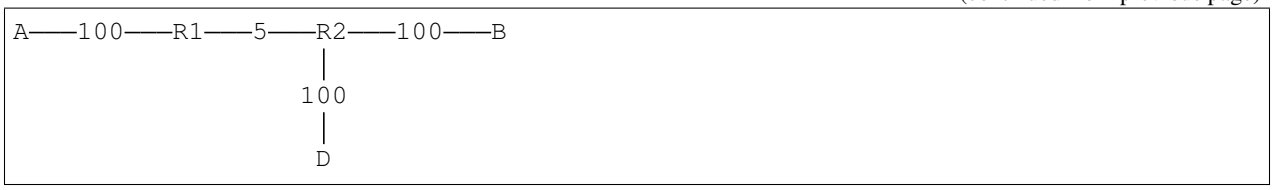
under the assumption that the bottleneck bandwidth was 1 packet per unit time. Give the formula when the bottleneck bandwidth is r packets per unit time. Hint: the formula above will apply if we measure time in units of 1/r; only the delays d, d_A and d_B need to be re-scaled to refer to "normal" time. A delay d measured in "normal" time corresponds to a delay $d' = r \times d$ measured in 1/r units.

2.0. Consider the following network, where the bandwidths marked are all in packets/ms. C is sending to D using sliding windows and A and B are idle.



(continues on next page)

(continued from previous page)



Suppose the one-way propagation delay on the 100 packet/ms links is 1 ms, and the one-way propagation delay on the R1–R2 link is 2 ms. The RTT_{noLoad} for the C–D path is thus about 8 ms, for a bandwidth×delay product of 40 packets. If C uses winsize = 50, then the queue at R1 will have size 10.

Now suppose A starts sending to B using sliding windows, also with winsize = 50. What will be the size of the queue at R1?

Hint: by symmetry, the queue will be equally divided between A's packets and C's, and A and C will each see a throughput of 2.5 packets/ms. RTT_{noLoad}, however, does not change. The number of packets in transit for each connection will be 2.5 packets/ms × RTT_{noLoad}.

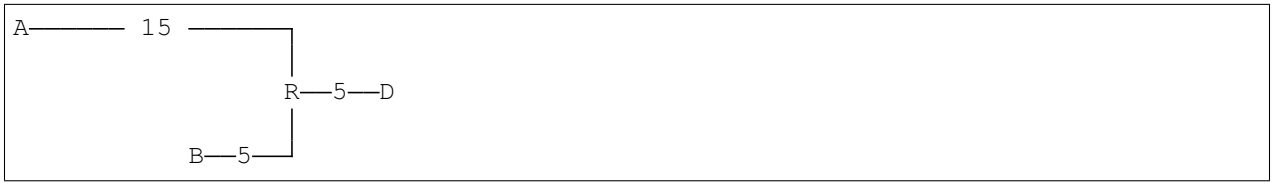
3.0. In the previous exercise, give the average number of **data** packets (not ACKs) in transit on each individual link:

(a).♢ for the original case in which C is the only sender, with winsize = 50 (the only active links here are C–R1, R1–R2 and R2–D).

(b). for the new case in which B is also sending, also with winsize = 50. In this case all links are active.

Each link will also have an equal number of **ACK** packets in transit in the reverse direction. Hint: since winsize ⩾ bandwidth×delay, packets are sent at the bottleneck rate.

4.0.♢ Consider the following network, with links labeled with one-way propagation delays in milliseconds (so, ignoring bandwidth delay, A's RTT_{noLoad} is 40 ms and B's is 20 ms). The bottleneck link is R–D, with a bandwidth of 6 packets/ms.



Initially B sends to D using a winsize of 120, the bandwidth×round-trip-delay product for the B–D path. A then begins sending as well, increasing its winsize until its share of the bandwidth is 2 packets/ms.

What is A's winsize at this point? How many packets do A and B each have in the queue at R?

It is perhaps easiest to solve this by repeated use of the observation that the number of packets in transit on a connection is always equal to RTT_{noLoad} times the actual bandwidth received by that connection. The algebraic methods of *20.2.3 Example 3: competition and queue utilization* can also be used, but bandwidth there was normalized to 1; all propagation delays given here would therefore need to be multiplied by 6.

5.0. Consider the C–D path from the diagram of *20.2.4 Example 4: cross traffic and RTT variation*:

```
C——100——R1——5——R2——100——D
```

Link numbers are bandwidths in packets/ms. Assume C is the only sender.

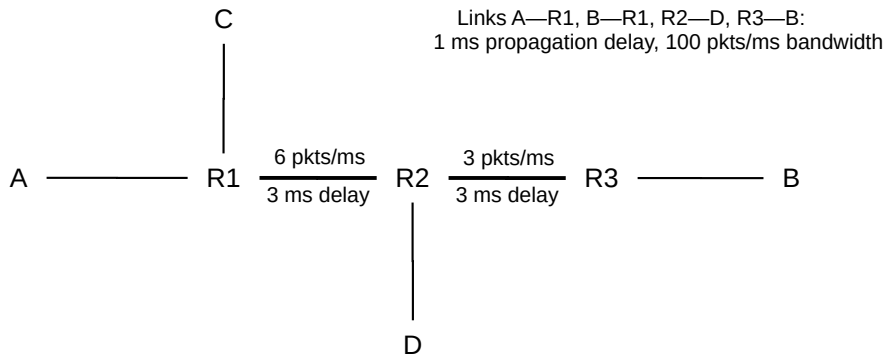
(a).♢ Give propagation delays for the links C–R1 and R2–D so that there will be an average of 5 packets in transit on the C–R1 and R2–D links, in each direction, if C uses a winsize sufficient to saturate the bottleneck R1–R2 link.

(b). Give propagation delays for all three links so that, when C uses a winsize equal to the round-trip transit capacity, there are 5 packets each way on the C–R1 link, 10 on the R1–R2 link, and 20 on the R2–D link.

6.0. Suppose we have the network layout below of *20.2.4 Example 4: cross traffic and RTT variation*, except that the R1–R2 bandwidth is 6 packets/ms and the R2–R3 bandwidth is 3 pkts/ms. The delays are as shown, making the C–D RTT_{noLoad} 10 ms and the A–B RTT_{noLoad} 16 ms. A connects to B and C connects to D.

(a).♢ Find window sizes w_A and w_C so that the A–B and C–D connections share the bottleneck R1–R2 bandwidth equally, and there is no queue.

(b). Show that increasing each of w_A and w_C by 30 packets leaves each connection with 30 packets in R1's queue – so the bandwidth is still shared equally – and none in R2's. Hint: As in (a), the A–B bandwidth cannot exceed 3 packets/ms, and C's packets can only accumulate at R1. To show A cannot have less than 50% of the bandwidth, observe that, if this happened, then A can have no queue at R2 (because packets now leave faster than they arrive), and so all of A's extra packets must also queue at R1.



7.0. Suppose we have the network layout of the previous exercise, 6.0. Suppose also that the A–B and C–D connections have settled upon window sizes as in 6.0(b), so that each contributes 30 packets to R1's queue. Each connection thus has 50% of the R1–R2 bandwidth and there is no queue at R2.

Now A's winsize is incremented by 10, initially, at least, leading to A contributing more than 50% of R1's queue. When the steady state is reached, how will these extra 10 packets be distributed between R1 and R2?

Hint: As A's winsize increases, A's overall throughput cannot rise due to the bandwidth restriction of the R2–R3 link.

8.0. Suppose we have the network layout of exercise 6.0, but modified so that the round-trip C–D RTT_{noLoad} is 5 ms. The round-trip A–B RTT_{noLoad} may be different.

The R1–R2 bandwidth is 6 packets/ms, so with A idle the C–D throughput is 6 packets/ms.

(a). Suppose that A and C have window sizes such that, with *both* transmitting, each has 30 packets in the queue at R1. What is C's winsize? Hint: C's throughput is now 3 packets/ms.
(b). Now suppose C's winsize, with A idle, is 60. In this case the C–D transit capacity would be 5 ms × 6 packets/ms = 30 packets, and so C would have 60–30 = 30 packets in R1's queue. A then begins sending, with a winsize chosen so that A and C's contributions to R1's queue are equal; C's winsize remains at 60. What will be C's (and thus A's) queue usage at R1? Hint: find the transit capacity for a throughput of 3 packets/ms.
(c). Suppose the A–B RTT_{noLoad} is 10 ms. If C's winsize is 60, find the winsize for A that makes A and C's contributions to R1's queue equal.

9.0. One way to address the reduced bandwidth TCP Reno gives to long-RTT connections is for all connections to use an increase increment of RTT^2 instead of 1; that is, everyone uses AIMD(RTT^2,1/2) instead of AIMD(1,1/2) (or AIMD(k×RTT^2,1/2), where k is an arbitrary scaling factor that applies to everyone).

(a). Construct a table in the style of of *20.3.2 Example 3: Longer RTT* above, showing the result of two connections using this strategy, where one connection has RTT = 1 and the other has RTT = 2. Start the connections with `cwnd`=RTT^2, and assume a loss occurs when $\texttt{cwnd}_1 + \texttt{cwnd}_2 > 24$.
(b). Explain why this strategy might not be desirable if one connection is over a direct LAN with an RTT of 1 ms, while the second connection has a very long path and an RTT of 1.0 sec. (Hint: the `cwnd`-increment value for the short-RTT connection would have to apply whether or not the long-RTT connection was present.)

10.0. Suppose two 1 kB packets are sent as part of a packet-pair probe, and the minimum time measured between arrivals is 5 ms. What is the estimated bottleneck bandwidth?

11.0. Consider the following three causes of a 1-second network delay between A and B. In all cases, assume ACKs travel instantly from B back to A.

> (i) An intermediate router with a 1-second-per-packet bandwidth delay; all other bandwidth delays negligible
>
> (ii) An intermediate link with a 1-second propagation delay; all bandwidth delays negligible
>
> (iii) An intermediate router with a 100-ms-per-packet bandwidth delay, and a steadily replenished queue of 10 packets, from another source (as in the diagram in *20.2.4 Example 4: cross traffic and RTT variation*).

(a). Suppose that, in each of these cases, the packet-pair technique (*20.2.6 Packet Pairs*) is used to measure the bandwidth. Assuming no packet reordering, what is the minimum time interval we could expect in each

case?

(b). What would be the corresponding values of the measured bandwidths, in packets per second? (For purposes of bandwidth measurement, you may assume that the "negligible" bandwidth delay in case (ii) is 0.01 sec.)

12.0. Suppose A sends packets to B using TCP Reno. The round-trip propagation delay is 1.0 seconds, and the bandwidth is 100 packets/sec (1 packet every 10 ms).

(a). Give RTT_{actual} when the window size has reached 100 packets.

(b). Give RTT_{actual} when the window size has reached 200 packets.

21 FURTHER DYNAMICS OF TCP

Is TCP Reno fair? Before we can ask that, we have to establish what we mean by fairness. We also look more carefully at the long-term behavior of TCP Reno (and Reno-like) connections, as the value of `cwnd` increases and decreases according to the TCP sawtooth. In particular we analyze the average `cwnd`; recall that the average `cwnd` divided by the RTT is the connection's average throughput (we momentarily ignore here the fact that RTT is not constant, but the error this introduces is usually small).

In the end, after establishing a fundamental relationship between TCP Reno `cwnd` and the packet loss rate, we end up declaring that maybe the best we can do is to assert that whatever TCP Reno does is "Reno fair", and establish a rule for "TCP [Reno] Friendliness".

The latter part of this chapter discusses "Active Queue Management": the idea that routers can make some assumptions about TCP traffic to better manage the flows passing through them. It turns out that routers can take advantage of TCP's behavior to provide better overall performance.

The chapter closes with the "high-bandwidth TCP problem" and related TCP issues.

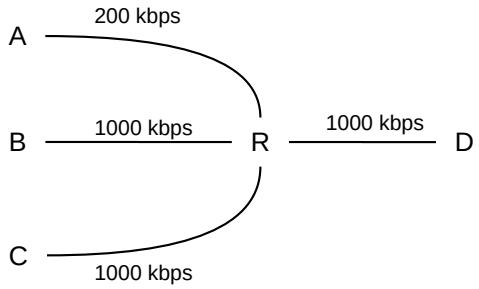
21.1 Notions of Fairness

There are several definitions for fair allocation of bandwidth among flows sharing a bottleneck link. One is **equal-shares fairness**; another is what we might call **TCP-Reno fairness**: to divide the bandwidth the way TCP Reno would. There are additional approaches to deciding what constitutes a fair allocation of bandwidth.

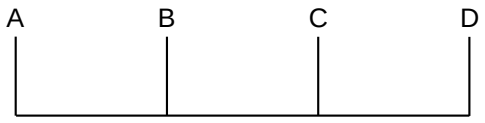
21.1.1 Max-Min Fairness

A natural generalization of equal-shares fairness to the case where some flows may be capped is **max-min fairness**, in which no flow bandwidth can be increased without decreasing some *smaller* flow rate. Alternatively, we maximize the bandwidth of the smallest-capacity flow, and then, with that flow fixed, maximize the flow with the next-smallest bandwidth, *etc*. A more intuitive explanation is that we distribute bandwidth in tiny increments equally among the flows, until the bandwidth is exhausted (meaning we have divided it equally), or one flow reaches its externally imposed bandwidth cap. At this point we continue incrementing among the remaining flows; any time we encounter a flow's external cap we are done with it.

As an example, consider the following, where we have connections A–D, B–D and C–D, and where the A–R link has a bandwidth of 200 kbps and all other links are 1000 kbps. Starting from zero, we increment the allocations of each of the three connections until we get to 200 kbps per connection, at which point the A–D connection has maxed out the capacity of the A–R link. We then continue allocating the remaining 400 kbps equally between B–D and C–D, so they each end up with 400 kbps.



As another example, known as the **parking-lot topology**, suppose we have the following network:



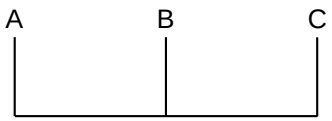
There are four connections: one from A to D covering all three links, and three single-link connections A–B, B–C and C–D. Each link has the same bandwidth. If bandwidth allocations are incrementally distributed among the four connections, then the first point at which any link bandwidth is maxed out occurs when all four connections each have 50% of the link bandwidth; max-min fairness here means that each connection has an equal share.

21.1.2 Proportional Fairness

A bandwidth allocation of rates $\langle r_1,r_2,\ldots,r_N\rangle$ for N connections satisfies **proportional fairness** if it is a legal allocation of bandwidth, and for any other allocation $\langle s_1,s_2,\ldots,s_N\rangle$, the aggregate proportional change satisfies

$$(r_1-s_1)/s_1 + (r_2-s_2)/s_2 + \ldots + (r_N-s_N)/s_N < 0$$

Alternatively, proportional fairness means that the sum $\log(r_1)+\log(r_2)+\ldots+\log(r_N)$ is minimized. If the connections share only the bottleneck link, proportional fairness is achieved with equal shares. However, consider the following two-stage parking-lot network:



Suppose the A–B and B–C links have bandwidth 1 unit, and we have three connections A–B, B–C and A–C. Then a proportionally fair solution is to give the A–C link a bandwidth of 1/3 and each of the A–B and B–C links a bandwidth of 2/3 (so each link has a total bandwidth of 1). For any change Δb in the bandwidth for the A–C link, the A–B and B–C links each change by $-\Delta b$. Equilibrium is achieved at the point where a 1% reduction in the A–C link results in two 0.5% increases, that is, the bandwidths are divided in proportion 1:2. Mathematically, if x is the throughput of the A–C connection, we are minimizing $\log(x) + 2\log(1-x)$.

Proportional fairness partially addresses the problem of TCP Reno's bias against long-RTT connections; specifically, TCP's bias here is still not proportionally fair, but TCP's response is closer to proportional fairness than it is to max-min fairness. See [HBT99].

21.2 TCP Reno loss rate versus `cwnd`

It turns out that we can express a connection's average `cwnd` in terms of the **packet loss rate**, p, *eg* p = 10^{-4} = one packet lost in 10,000. The relationship comes by assuming that all packet losses are because the network ceiling was reached. We will also assume that, when the network ceiling is reached, only one packet is lost, although we can dispense with this by counting a "cluster" of related losses (within, say, one RTT) as a single *loss event*.

Let C represent the network ceiling – so that when `cwnd` reaches C a packet loss occurs. While C is constant only for a very stable network, C usually does not vary by much; we will assume here that it is constant. Then `cwnd` varies between C/2 and C, with packet drops occurring whenever `cwnd` = C is reached. Let N = C/2. Then between two consecutive packet loss events, that is, over one "tooth" of the TCP connection, a total of N+(N+1)+ ... +2N packets are sent in N+1 flights; this sum can be expressed algebraically as 3/2 N(N+1) $\simeq$ 1.5 N^2. The loss rate is thus one packet out of every 1.5 N^2, and the loss rate p is $1/(1.5\ N^2)$.

The average `cwnd` in this scenario is 3/2 N (that is, the average of N=$\texttt{cwnd}_{min}$ and 2N=$\texttt{cwnd}_{max}$). If we let M = 3/2 N represent the average `cwnd`, $\texttt{cwnd}_{mean}$, we can express the above loss rate in terms of M: the number of packets between losses is 2/3 M^2, and so p=3/2 M^{-2}.

Now let us solve this for M=$\texttt{cwnd}_{mean}$ in terms of p; we get $M^2 = 3/2\ p^{-1}$ and thus

$$M = \texttt{cwnd}_{mean} = 1.225\ p^{-1/2}$$

where 1.225 is the square root of 3/2. Seen in this form, a given network loss rate sets the window size; this loss rate is ultimately tied to the network capacity. If we are interested in the maximum `cwnd` instead of the mean, we multiply the above by 4/3.

From the above, the *bandwidth* available to a connection is now as follows (though RTT may not be constant):

$$\text{bandwidth} = \texttt{cwnd}/\text{RTT} = 1.225/(\text{RTT} \times \sqrt{p})$$

In [PFTK98] the authors consider a TCP Reno model that takes into account the measured frequency of coarse timeouts (in addition to fast-recovery responses leading to `cwnd` halving), and develop a related formula with additional terms.

As the bottleneck queue capacity increases, both `cwnd` and the number of packets between losses (1/p) increase, connected as above. Once the queue is large enough that the bottleneck link is 100% utilized, however, the bandwidth no longer increases.

Another way to view this formula is to recall that 1/p is the number of packets per tooth; that is, 1/p is the tooth "area". Squaring both sides, the formula says that the TCP Reno tooth area is proportional to the square of the average tooth height (that is, to $\texttt{cwnd}_{mean}$) as the network capacity increases (that is, as $\texttt{cwnd}_{mean}$ increases).

21.2.1 Irregular teeth

In the preceding, we assumed that all teeth were the same size. What if they are not? In [OKM96], this problem was considered under the assumption that every packet faces the same (small) loss probability (and so the intervals between packet losses are exponentially distributed). In this model, it turns out that the above formula still holds except the constant changes from 1.225 to 1.309833.

To understand how irregular teeth lead to a bigger constant, imagine sending a large number K of packets which encounter n losses. If the losses are regularly spaced, then the TCP graph will have n equally sized teeth, each with K/n packets. But if the n losses are randomly distributed, some teeth will be larger and some will be smaller. The *average* tooth height will be the same as in the regularly-spaced case (see exercise 7.0). However, the number of packets in any one tooth is generally related to the *square* of the height of that tooth, and so larger teeth will count disproportionately more. Thus, the random distribution will have a higher total number of packets delivered and thus a higher mean `cwnd`.

See also exercise 17.0, for a simple simulation that generates a numeric estimate for the constant 1.309833.

Note that losses at uniformly distributed random intervals may not be an ideal model for TCP either; in the presence of congestion, loss events are far from statistical independence. In particular, immediately following one loss another loss is unlikely to occur until the queue has time to fill up.

21.2.2 Unsynchronized TCP Losses

In *20.3.3 TCP Reno RTT bias* we considered a model in which all loss events are fully synchronized; that is, whenever the queue becomes full, *both* TCP Reno connections always experience packet loss. In that model, if $RTT_2/RTT_1 = \lambda$ then $\texttt{cwnd}_1/\texttt{cwnd}_2 = \lambda$ and $\text{bandwidth}_1/\text{bandwidth}_2 = \lambda^2$, where $\texttt{cwnd}_1$ and $\texttt{cwnd}_2$ are the respective average values for `cwnd`.

What happens if loss events for two connections do not have such a neat one-to-one correspondence? We will derive the ratio of loss events (or, more precisely, TCP loss *responses*) for connection 1 versus connection 2 in terms of the bandwidth and RTT ratios, without using the synchronized-loss hypothesis.

Note that we are comparing the total number of loss events (or loss responses) here – the total number of TCP Reno teeth – over a large time interval, and not the relative *per-packet* loss probabilities. One connection might have numerically more losses than a second connection but, by dint of a smaller RTT, send more packets between its losses than the other connection and thus have *fewer* losses per packet.

Let losscount_1 and losscount_2 be the number of loss responses for each connection over a long time interval T. For i=1 and i=2, the i^{th} connection's per-packet loss probability is $p_i = \text{losscount}_i/(\text{bandwidth}_i \times T) = (\text{losscount}_i \times RTT_i)/(\texttt{cwnd}_i \times T)$. But by the result of *21.2 TCP Reno loss rate versus cwnd*, we also have $\texttt{cwnd}_i = k/\sqrt{p_i}$, or $p_i = k^2/\texttt{cwnd}_i^2$. Equating, we get

$$p_i = k^2/\texttt{cwnd}_i^2 = (\text{losscount}_i \times RTT_i) \,/\, (\texttt{cwnd}_i \times T)$$

and so

$$\text{losscount}_i = k^2 T \,/\, (\texttt{cwnd}_i \times RTT_i)$$

Dividing and canceling, we get

$$\text{losscount}_1/\text{losscount}_2 = (\texttt{cwnd}_2/\texttt{cwnd}_1) \times (RTT_2/RTT_1)$$

We will make use of this in *31.4.2.2 Relative loss rates*.

We can go just a little further with this: let γ denote the losscount ratio above:

$\gamma = (\texttt{cwnd}_2/\texttt{cwnd}_1) \times (\text{RTT}_2/\text{RTT}_1)$

Therefore, as $\text{RTT}_2/\text{RTT}_1 = \lambda$, we must have $\texttt{cwnd}_2/\texttt{cwnd}_1 = \gamma/\lambda$ and thus

$\text{bandwidth}_1/\text{bandwidth}_2 = (\texttt{cwnd}_1/\texttt{cwnd}_2) \times (\text{RTT}_2/\text{RTT}_1) = \lambda^2/\gamma$.

Note that if γ=λ, that is, if the longer-RTT connection has fewer loss events in exact inverse proportion to the RTT, then $\text{bandwidth}_1/\text{bandwidth}_2 = \lambda = \text{RTT}_2/\text{RTT}_1$, and also $\texttt{cwnd}_1/\texttt{cwnd}_2 = 1$.

21.3 TCP Friendliness

Suppose we are sending packets using a non-TCP real-time protocol. How are we to manage congestion? In particular, how are we to manage congestion in a way that treats other connections – particularly TCP Reno connections – fairly?

For example, suppose we are sending interactive audio data in a congested environment. Because of the real-time nature of the data, we cannot wait for lost-packet recovery, and so must use UDP rather than TCP. We might further suppose that we can modify the encoding so as to reduce the sending rate as necessary – that is, that we are using *adaptive* encoding – but that we would prefer in the absence of congestion to keep the sending rate at the high end. We might also want a relatively uniform *rate* of sending; the TCP sawtooth leads to periodic variations in throughput that we may wish to avoid.

Our application may not be windows-based, but we can still monitor the number of packets it has in flight on the network at any one time; if the packets are small, we can count bytes instead. We can use this count instead of the TCP `cwnd`.

We will say that a given communications strategy is **TCP Friendly** if the number of packets on the network at any one time is approximately equal to the TCP Reno $\texttt{cwnd}_{\text{mean}}$ for the prevailing packet loss rate p. Note that – assuming losses are independent events, which is definitely not quite right but which is often Close Enough – in a long-enough time interval, all connections sharing a common bottleneck can be expected to experience approximately the same packet loss rate.

The point of TCP Friendliness is to regulate the number of the non-Reno connection's outstanding packets in the presence of competition with TCP Reno, so as to achieve a degree of fairness. In the absence of competition, the number of any connection's outstanding packets will be bounded by the transit capacity plus capacity of the bottleneck queue. Some non-Reno protocols (*eg* TCP Vegas, *22.6 TCP Vegas*, or constant-**rate** traffic, *21.3.2 RTP*) may in the absence of competition have a loss rate of zero, simply because they never overflow the queue.

Another way to approach TCP Friendliness is to start by *defining* "Reno Fairness" to be the bandwidth allocations that TCP Reno assigns in the face of competition. TCP Friendliness then simply means that the given non-Reno connection will get its Reno-Fair share – not more, not less.

We will return to TCP Friendliness in the context of general AIMD in *21.4 AIMD Revisited*.

21.3.1 TFRC

TFRC, or TCP-Friendly Rate Control, RFC 3448, uses the loss rate experienced, p, and the formulas above to calculate a sending rate. It then allows sending at that rate; that is, TFRC is rate-based rather than window-based. As the loss rate increases, the sending rate is adjusted downwards, and so on. However, adjustments are done more smoothly than with TCP, giving the application a more gradually changing transmission rate.

From RFC 5348:

> TFRC is designed to be reasonably fair when competing for bandwidth with TCP flows, where we call a flow "reasonably fair" if its sending rate is generally within a **factor of two** of the sending rate of a TCP flow under the same conditions. [emphasis added; a factor of two might not be considered "close enough" in some cases.]

The penalty of having smoother throughput than TCP while competing fairly for bandwidth is that TFRC responds more slowly than TCP to changes in available bandwidth.

TFRC senders include in each packet a sequence number, a timestamp, and an estimated RTT.

The TFRC receiver is charged with sending back feedback packets, which serve as (partial) acknowledgments, and also include a receiver-calculated value for the loss rate over the previous RTT. The response packets also include information on the current actual RTT, which the sender can use to update its estimated RTT. The TFRC receiver might send back only one such packet per RTT.

The actual response protocol has several parts, but if the loss rate increases, then the primary feedback mechanism is to *calculate* a new (lower) sending rate, using some variant of the `cwnd` = k/$\sqrt{}$p formula, and then shift to that new rate. The rate would be cut in half only if the loss rate p quadrupled.

Newer versions of TFRC have a various features for responding more promptly to an unusually sudden problem, but in normal use the calculated sending rate is used most of the time.

21.3.2 RTP

The **Real-Time Protocol**, or RTP, is sometimes (though not always) coupled with TFRC. RTP is a UDP-based protocol for streaming time-sensitive data.

Some RTP features include:

- The sender establishes a *rate* (rather than a window size) for sending packets
- The receiver returns periodic summaries of loss rates
- ACKs are relatively infrequent
- RTP is suitable for *multicast* use; a very limited ACK rate is important when every packet sent might have hundreds of recipients
- The sender adjusts its `cwnd`-equivalent up or down based on the loss rate and the TCP-friendly `cwnd`=k/$\sqrt{}$p rule
- Usually some sort of "stability" rule is incorporated to avoid sudden changes in rate

As a common RTP example, a typical VoIP connection using a DS0 (64 kbps) rate might send one packet every 20 ms, containing 160 bytes of voice data, plus headers.

For a combination of RTP and TFRC to be useful, the underlying application must be **rate-adaptive**, so that the application can still function when the available rate is reduced. This is often not the case for simple VoIP encodings; see *25.11.4 RTP and VoIP*.

We will return to RTP in *25.11 Real-time Transport Protocol (RTP)*.

The UDP-based QUIC transport protocol (*16.1.1 QUIC*) uses a congestion-control mechanism compatible with Cubic TCP (*22.15 TCP CUBIC*), which isn't quite the same as TCP Reno. But QUIC could just as easily have used TFRC to achieve TCP-Reno-friendliness.

21.3.3 DCCP Congestion Control

We saw DCCP earlier in *16.1.2 DCCP* and *18.15.3 DCCP*. DCCP also includes a set of congestion-management "profiles"; a connection can choose the profile that best fits its needs. The two standard ones are the TCP-Reno-like profile (**RFC 4341**) and the TFRC profile (**RFC 4342**).

In the Reno-like profile, every packet is acknowledged (though, as with TCP, ACKs may be sent on the arrival of every other Data packet). Although DCCP ACKs are not cumulative, use of the TCP-SACK-like ACK-vector format ensures that acknowledgments are received reliably except in extreme-loss situations.

The sender maintains `cwnd` much as a TCP Reno sender would. It is incremented by one for each RTT with no loss, and halved in the event of packet loss. Because sliding windows is not used, `cwnd` does not represent a window size. Instead, the sender maintains an Estimated FlightSize (*19.4 TCP Reno and Fast Recovery*), which is the sender's best guess at the number of outstanding packets. In **RFC 4341** this is referred to as the **pipe value**. The sender is then allowed to send additional packets as long as `pipe` < `cwnd`.

The Reno-like profile also includes a slow start mechanism.

In the TFRC profile, an ACK is sent at least once per RTT. Because ACKs are sent less frequently, it may occasionally be necessary for the sender to send an ACK of ACK.

As with TFRC generally, a DCCP sender using the TFRC profile has its rate limited, rather than its window size.

DCCP provides a convenient programming framework for use of TFRC, complete with (at least in the Linux world), a traditional socket interface. The developer does not have to deal with the TFRC rate calculations directly.

21.4 AIMD Revisited

TCP Tahoe chose an increase increment of 1 on no losses, and a decrease factor of 1/2 otherwise.

Another approach to TCP Friendliness is to retain TCP's additive-increase, multiplicative-decrease strategy, but to change the numbers. Suppose we denote by AIMD(α,β) the strategy of incrementing the window size by α after a window of no losses, and multiplying the window size by (1-β)<1 on loss (so β=0.1 means the window is reduced *by* 10%). TCP Reno is thus AIMD(1,0.5).

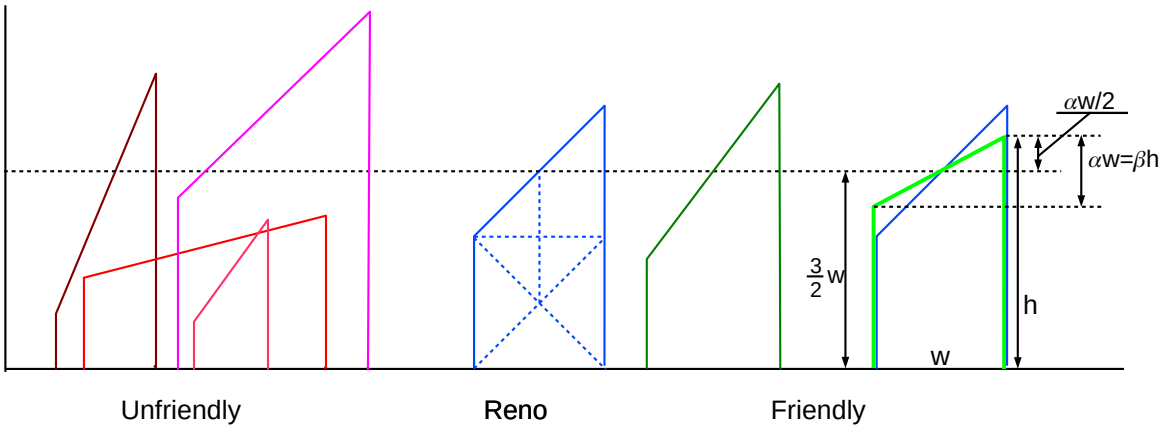
Any AIMD(α,β) protocol also follows a sawtooth, where the slanted top to the tooth has slope α. All combinations of α>0 and 0<β<1 are possible. The dimensions of one tooth of the sawtooth are somewhat constrained by α and β. Let h be the maximum height of the tooth and let w be the width (as measured in

RTTs). Then, if the losses occur at regular intervals, the height of the tooth at the left (low) edge is (1-β)h and the total vertical difference is βh. This vertical difference must also be αw, and so we get αw = βh, or h/w = α/β; these values are labeled on the rightmost teeth in the diagram below. These equations mean that the proportions of the tooth (h to w) are determined by α and β. Finally, the mean height of the tooth is (1-β/2)h.

We are primarily interested in AIMD(α,β) cases which are TCP Friendly (*21.3 TCP Friendliness*). TCP friendliness means that an AIMD(α,β) connection with the same loss rate as TCP Reno will have the same mean `cwnd`. Each tooth of the sawtooth represents one loss. The number of packets sent per tooth is, using h and w as in the previous paragraph, (1-β/2)hw.

Geometrically, the number of packets sent per tooth is the *area* of the tooth, so two connections with the same per-packet loss rate will have teeth with the same area. TCP Friendliness means that two connections will have the same mean `cwnd` and thus the same average tooth height. If the teeth of two connections have the same area and the same average height, they must have the same width (in RTTs), and thus that the rates of loss per unit *time* must be equal, not just the rates of loss per number of packets.

The diagram below shows a TCP Reno tooth (blue) together with some unfriendly AIMD(α,β) teeth on the left (red) and two friendly teeth on the right (green), the second friendly tooth is superimposed on the Reno tooth.



AIMD teeth; Reno tooth has guide-lines showing midpoints

The additional dashed lines within the central Reno tooth demonstrate the Reno $1\times1\times2$ proportions, and show that the horizontal dashed line, representing $\texttt{cwnd}_{\text{mean}}$, is at height 3/2 w, where w is, as before, the width.

In the rightmost green tooth, superimposed on the Reno tooth, we can see that h = (3/2)$\times$w + (α/2)$\times$w. We already know h = (α/β)$\times$w; setting these expressions equal, canceling the w and multiplying by 2 we get (3+α) = 2α/β, or β = 2α/(3+α). Solving for β we get

$$\alpha = 3\beta/(2-\beta)$$

or $\alpha \simeq 1.5\beta$ for small β. As the reduction factor 1-β gets closer to 1, the protocol can remain TCP-friendly by appropriately reducing α; eg AIMD(1/5, 1/8).

Having a small β means that a connection does not have sudden bandwidth drops when losses occur; this can be important for applications that rely on a regular rate of data transfer (such as voice). Such applications are sometimes said to be slowly responsive, in contrast to TCP's `cwnd` = `cwnd`/2 rapid response.

21.4.1 AIMD and Convergence to Fairness

While TCP-friendly AIMD(α,β) protocols will converge to fairness when competing with TCP Reno (with equal RTTs), a consequence of decreasing β is that fairness may take longer to arrive; here is an example. We will assume, as above in *20.3.3 TCP Reno RTT bias*, that loss events for the two competing connections are synchronized. Recall that for two same-RTT TCP Reno connections (that is, AIMD(α,β) where β=1/2), if the initial difference in the connections' respective `cwnds` is D, then D is reduced by half on each loss event.

Now suppose we have two AIMD(α,β) connections with some other value of β, and again with a difference D in their `cwnd` values. The two connections will each increase `cwnd` by α each RTT, and so when losses are not occurring D will remain constant. At loss events, D will be reduced by a factor of 1-β. If β=1/4, corresponding to α=3/7, then at each loss event D will be reduced only to 3/4 D, and the "half-life" of D will be almost twice as large. The two connections will still converge to fairness as D→0, but it will take twice as long.

21.5 Active Queue Management

Active Queue Management (AQM) means that routers take some active steps to manage their queues. The primary goal of AQM is to reduce excessive queuing delays; *cf 21.5.1 Bufferbloat*. A secondary goal is to improve the performance of TCP connections – which constitute the vast majority of Internet traffic – through the router. By signaling to TCP connections that they should reduce `cwnd`, overall queuing delays are also reduced.

Generally routers manage their queues either by **marking** packets or by **dropping** them. All routers drop packets when there is no more space for new arrivals, but this falls into the category of active management when packets are dropped before the queue has run completely out of space. Queue management can be done at the congestion "knee", when queues just start to build (and when marking is more appropriate), or as the queue starts to become full and approaches the "cliff".

Broadly speaking, the priority queuing and random drop mechanisms (*20.1 A First Look At Queuing*) might be considered forms of AQM, at least if the goal was to manage the overall queue size. So might fair queuing and hierarchical queuing (*23 Queuing and Scheduling*). The mechanisms most commonly associated with the AQM category, though, are RED, below, and its successors, especially CoDel. For a discussion of the potential benefits of fair queuing to queue management, see *23.6.1 Fair Queuing and Bufferbloat*.

21.5.1 Bufferbloat

As we saw in *19.7 TCP and Bottleneck Link Utilization*, TCP Reno connections are happiest when the queue capacity at the bottleneck router exceeds the bandwidth × delay transit capacity. But it is easy to get carried away here. The calculations of *19.7.1 TCP Queue Sizes* suggested that an optimum backbone-router buffer size for TCP Reno might be hundreds of megabytes. Because RAM is cheap, and because more space is hard to say no to, queue sizes in the real world often tend to be at the larger end of the scale. Excessive delay due to excessive queue capacity is known as **bufferbloat**. Of course, "excessive" is a matter of perspective; if the only traffic you're interested in is bulk TCP flows, large queues are good. But if you're interested in real-time traffic like voice and interactive video, or even simply in fast web-page loads, bufferbloat becomes a problem. Large queues can also lead to delay *variability*, or jitter.

Backbone routers are one class of offender here, but not the only. Many residential routers have a queue capacity several times the average bandwidth × delay product, meaning that queuing delay potentially becomes much larger than propagation delay. Even end-systems often have large queues; on Linux systems, the default queue size can be several hundred packets.

All these delay-related issues do not play well with interactive traffic or real-time traffic (**RFC 7567**). As a result, there are proposals for running routers with much smaller queues; see, for example, [WM05] and [EGMR05]. This may reduce the bottleneck link utilization of a *single* TCP flow to 75%. However, with *multiple* TCP flows having unsynchronized losses, the situation will often be much better.

Still, for router managers, deciding on a queue capacity can be a vexing issue. The CoDel algorithm, below, offers great promise, but we start with some earlier strategies.

21.5.2 DECbit

In the congestion-avoidance technique proposed in [RJ90], routers encountering early signs of congestion **marked** the packets they forwarded; senders used these markings to adjust their window size. The system became known as DECbit in reference to the authors' employer and was implemented in DECnet (closely related to the OSI protocol suite), though apparently there was never a TCP/IP implementation. The idea behind DECbit eventually made it into TCP/IP in the form of ECN, below, but while ECN – like TCP's other congestion responses – applies control near the congestion cliff, DECbit proposed introducing control when congestion was still minimal, just above the congestion knee. DECbit was never a solution to bufferbloat; in the DECbit era, memory was expensive and queue capacities were seldom excessive.

The DECbit mechanism allowed routers to set a designated "congestion bit". This would be set in the data packet being forwarded, but the status of this bit would be echoed back in the corresponding ACK (otherwise the sender would never hear about the congestion).

DECbit *routers* defined "congestion" as an average queue size greater than 1.0; that is, congestion meant that the connection was just past the "knee". Routers would set the congestion bit whenever this average-queue condition was met.

The target for DECbit *senders* would then be to have 50% of packets marked as "congested". If fewer than 50% of packets were marked, `cwnd` would be incremented by 1; if more than 50% were marked, then `cwnd` would be decreased by a factor of 0.875. Note this is very different from the TCP approach in that DECbit begins marking packets at the congestion "knee" while TCP Reno responds only to packet losses which occur at the "cliff".

A consequence of this knee-based mechanism is that DECbit shoots for very limited queue utilization, unlike TCP Reno. At a congested router, a DECbit connection would attempt to keep about 1.0 packets in the router's queue, while a TCP Reno connection might fill the remainder of the queue. Thus, DECbit would in principle compete poorly with any connection where the sender ignored the marked packets and simply tried to keep `cwnd` as large as possible. As we will see in *22.6 TCP Vegas*, TCP Vegas also strives for limited queue utilization; in *31.5 TCP Reno versus TCP Vegas* we investigate through simulation how fairly TCP Vegas competes with TCP Reno.

21.5.3 Explicit Congestion Notification (ECN)

ECN is the TCP/IP equivalent of DECbit, though the actual mechanics are quite different. The current version is specified in **RFC 3168**, modifying an earlier version in **RFC 2481**. The IP header contains a two-bit ECN field, consisting of the ECN-Capable Transport (ECT) bit and the Congestion Experienced (CE) bit; the ECN field is shown in *9.1 The IPv4 Header*. The ECT bit is set by a sender to indicate to routers that it is able to use the ECN mechanism. (These are actually the older **RFC 2481** names for the bits, but they will serve our purposes here.) The TCP header contains an additional two bits: the ECN-Echo bit (ECE) and the Congestion Window Reduced (CWR) bit; these are shown in the fourth row in *17.2 TCP Header*.

The original goal of ECN was to improve TCP throughput by eliminating most actual packet losses and the resultant timeouts. Bufferbloat was, again, not at issue.

ECN and Middleboxes

In the early days of ECN, some non-ECN-aware firewalls responded to packets with the CWR or ECE bits set by dropping them and returning a spoofed RST packet to the sender. See **RFC 3360** for details and a discussion of this nonstandard use of RST. This is a good example of how "middleboxes" are sometimes obstacles to protocol evolution; see *9.7.2 Middleboxes*.

Routers set the CE bit in the IP header when they might otherwise drop the packet (or possibly when the queue is at least half full, or in lieu of a RED drop, below). As in DECbit, receivers echo the CE status back to the sender in the ECE bit of the next ACK; the reason for using the ECE bit is that this bit belongs to the TCP header and thus the TCP layer can be assured of control of it.

TCP senders treat ACKs with the ECE bit set the same as if a loss occurred: `cwnd` is cut in half. Because there is no actual loss, the arriving ACKs can still pace continued sliding-windows sending. The Fast Recovery mechanism is not needed.

When the TCP sender has responded to an ECE bit (by halving `cwnd`), it sets the CWR bit. Once the receiver has received a packet with the CE bit set in the IP layer, it sets the ECE bit in all subsequent ACKs until it receives a data packet with the CWR bit set. This provides for reliable communication of the congestion information, and helps the sender respond just once to multiple packet losses within a single windowful.

Note that the initial packet marking is done at the IP layer, but the generation of the marked ACK and the sender response to marked packets is at the TCP layer (the same is true of DECbit though the layers have different names).

Only a packet that would otherwise have been dropped has its CE bit set; the router does *not* mark all waiting packets once its queue reaches a certain threshold. Any marked packet must, as usual, wait in the queue for its turn to be forwarded. The sender finds out about the congestion after one full RTT, versus one full RTT plus four packet transmission times for Fast Retransmit. A much earlier, "legacy" strategy was to require routers, upon dropping a packet, to immediately send back to the sender an `ICMP Source Quench` packet. This is a faster way (the fastest possible way) to notify a sender of a loss. It was never widely implemented, however, and was officially deprecated by **RFC 6633**.

Because ECN congestion is treated the same way as packet drops, ECN competes fairly with TCP Reno.

RFC 3540 is a proposal (as of 2016 not yet official) to slightly amend the mechanism described above to

support detection of receivers who attempt to conceal evidence of congestion. A receiver would do this by not setting the ECE bit in the ACK when a data packet arrives marked as having experienced congestion. Such an unscrupulous (or incorrectly implemented) receiver may then gain a greater share of the bandwidth, because its sender maintains a larger `cwnd` than it should. The amendment also detects erasure of the ECE bit (or other ECN bits) by middleboxes.

The new strategy, known as the **ECN nonce**, treats the ECN bits ECT and CE as a single unit. The value 00 is used by non-ECN-aware senders, and the value 11 is used by routers as the congestion marker. ECN-aware senders mark data packets by *randomly* choosing 10 (known as ECT(0)) or 01 (known as ECT(1)). This choice encodes the *nonce bit*, with ECT(0) representing a nonce bit of 0 and ECT(1) representing 1; the nonce bit can also be viewed as the value of the second ECN bit.

The receiver is now expected, in addition to setting the ECE bit, to also return the one-bit running sum of the nonce bits in a new TCP-header bit called the nonce-sum (NS) bit, which immediately precedes the CRW bit. This sum is over all data packets received since the previous packet loss or congestion-experienced packet. The point of this is that if the receiver attempts to conceal congestion by leaving the ECE bit zero, the receiver cannot properly set the NS bit, because it does not know which of ECT(0) or ECT(1) was used. For each packet marked as experiencing congestion, the receiver has a 50% chance of guessing correctly, but over time successful guessing becomes increasingly unlikely. If ECN-noncompliance is detected, the sender must now stop using ECN, and may choose a smaller `cwnd` as a precaution.

Although we have described ECN as a mechanism implemented by routers, it can just as easily be implemented by switches, and is available in many commercial switches.

21.5.4 RED

"Traditional" routers drop packets only when the queue is full; senders have no overt indication before then that the cliff is looming. ECN improves this by informing TCP connections of the impending cliff so they can reduce `cwnd` without actually losing packets. The idea behind **Random Early Detection** (RED) routers, introduced in [FJ93], is that the router is allowed to drop an occasional packet much earlier, say when the queue is less than half full. These early packet drops provide a signal to senders that they should slow down; we will call them **signaling losses**. While packets are indeed lost, they are dropped in such a manner that usually only one packet per windowful (per connection) will be lost. Classic TCP Reno, in particular, behaves poorly with multiple losses per window and RED is able to avoid such multiple losses. The primary goal of RED was, as with ECN, to improve TCP performance. Note that RED preceded ECN by six years.

RED is, strictly speaking, a *queuing discipline* in the sense of *23.4 Queuing Disciplines*; FIFO is another. It is often more helpful, however, to think of RED as a technique that an otherwise-FIFO router can use to improve the performance of TCP traffic through it.

Designing an early-drop algorithm is not trivial. A predecessor of RED known as Early Random Drop (ERD) gateways simply introduced a small uniform drop probability p, *eg* p=0.01, once the queue had reached a certain threshold. This addresses the TCP Reno issue reasonably well, except that dropping with a uniform probability p leads to a surprisingly high rate of multiple drops in a cluster, or of long stretches with no drops. More uniformity was needed, but drops at regular intervals are too uniform.

The actual RED algorithm does two things. First, the base drop probability – p_{base} – rises steadily from a minimum queue threshold q_{min} to a maximum queue threshold q_{max} (these might be 40% and 80% respectively of the absolute queue capacity); at the maximum threshold, the drop probability is still quite small.

The base probability p_{base} increases linearly in this range according to the following formula, where p_{max} is the maximum RED-drop probability; the value for p_{max} proposed in [FJ93] was 0.02.

$$p_{base} = p_{max} \times (avg\_queuesize - q_{min})/(q_{max} - q_{min})$$

Second, as time passes after a RED drop, the actual drop probability p_{actual} begins to rise, according to the next formula:

$$p_{actual} = p_{base} / (1 - count \times p_{base})$$

Here, count is the number of packets sent since the last RED drop. With count=0 we have $p_{actual} = p_{base}$, but p_{actual} rises from then on with a RED drop guaranteed within the next $1/p_{base}$ packets. This provides a mechanism by which RED drops are uniformly enough spaced that it is unlikely two will occur in the same window of the same connection, and yet random enough that it is unlikely that the RED drops will remain synchronized with a single connection, thus targeting it unfairly.

A significant drawback to RED is that the choice of the various parameters is decidedly *ad hoc*. It is not clear how to set them so that TCP connections with both small and large bandwidth×delay products are handled appropriately, or even how to set them for a given output bandwidth. The probability p_{base} should, for example, be roughly 1/winsize, but winsize for TCP connections can vary by several orders of magnitude. **RFC 2309**, from 1998, recommended RED, but its successor **RFC 7567** from 2015 has backed away from this, recommending instead that an appropriate AQM strategy be implemented, but that the details should be left to the discretion of the router manager.

In *25.8 RED with In and Out* we will look at an application of RED to quality-of-service guarantees.

21.5.5 ADT

The paper [SKS06] proposes the Adaptive Drop-Tail algorithm, in which the maximum queue capacity is adjusted at intervals (of perhaps 5 minutes) in order to maintain a specific desired link-utilization target (perhaps 95%). At the end of each interval, the available queue capacity is increased or decreased (perhaps by 5%) depending on whether the link utilization was under or over the target. ADT does not selectively drop packets otherwise; if there is space for an arriving packet within the current queue capacity, it is accepted.

ADT does a good job adjusting the overall queue capacity to meet circumstances that change slowly; an example might be the size of the user pool. However, ADT does not respond to short-term fluctuations. In particular, it does not attempt to respond to fluctuations occurring within a single TCP Reno tooth. ADT also does not maintain additional queue space for transient packet bursts.

21.5.6 CoDel

The CoDel queue-management algorithm (pronounced "coddle") attempts, like RED, to use signaling losses to encourage connections to reduce their queue utilization. This allows CoDel to maintain a large total queue capacity, available to absorb bursts, while at the same time maintaining on average a much smaller level of actual queue utilization. To achieve this, CoDel is able to distinguish between transient queue spikes and "standing-queue" utilization, persisting over multiple RTTs; the canonical example of the latter is TCP Reno's queue buildup towards the right-hand edge of each sawtooth. Unlike RED, CoDel has essentially no tunable parameters, and adapts to a wide range of bandwidths and traffic types. See [NJ12] and the Internet Draft draft-ietf-aqm-codel-06. Reducing bufferbloat is an explict goal of CoDel.

CoDel measures the minimum value of queue utilization over a designated short time period known as the **Interval**. The Interval is intended to be a little larger than most connection RTT_{noLoad} values; it is typically 100 ms. Through this minimum-utilization statistic, CoDel will easily be able to detect a TCP Reno connection's queue-building phase, which except for short-RTT connections will last for many Intervals.

CoDel measures this queue utilization in terms of the time the packet spends in the queue (its "sojourn time") rather than the size of the queue in bytes. While these two measures are proportional at any one router, the use of time rather than space means that the CoDel algorithm is independent of the outbound bandwidth, and so does not need to be configured for that bandwidth.

CoDel's target for the minimum queue utilization is typically 5% of the interval, or 5 ms, although 10% is also reasonable. If the minimum utilization is smaller, no action is taken. If the minimum utilization becomes larger, then CoDel enters its "dropping mode", drops a packet, and begins scheduling additional packet drops. This lasts until the minimum utilization is again below the target, and CoDel returns to its "normal mode".

Once dropping mode begins, the second drop is scheduled for one Interval after the first drop, though the second drop may not occur if CoDel is able to return to normal mode. While CoDel remains in dropping mode, additional packet drops are scheduled after times of Interval/$\sqrt{2}$, Interval/$\sqrt{3}$, *etc*; that is, the dropping rate accelerates in proportion to $\sqrt{n}$ until the minimum time packets spend in the queue is small enough again that CoDel is able to return to normal mode.

If the traffic consists of a single TCP Reno connection, CoDel will drop one of its packets as soon as the queue utilization hits 5%. That will cause `cwnd` to halve, most likely making even a second packet drop unnecessary. If the connection's RTT was approximately equal to the Interval, then its link utilization will be the same as if the queue capacity was fixed at 5% of the transit capacity, or 79% (*19.12 Exercises*, 13.0). However, if there are a modest number of unsynchronized TCP connections, the link-utilization rate climbs to above 90% ([NJ12], figs 5 and 8).

If the traffic consists of several TCP Reno connections, a few drops should be all that are necessary to force most of the connections to halve their `cwnd`s, and thus greatly reduce their collective queue utilization. However, even if the traffic consists of a single *fixed-rate* UDP connection, with too high a rate for the bottleneck, CoDel still works. In this case it drops as many packets as it needs to in order to drive down the queue utilization, and this cycle repeats as necessary. An additional feature of CoDel is that if the dropping mode is re-entered quickly, the dropping rate picks up where it left off.

For an example application of CoDel, see *24.6 Limiting Delay*.

21.6 The High-Bandwidth TCP Problem

The TCP Reno algorithm has a serious consequence for high-bandwidth connections: the `cwnd` needed implies a very small – unrealistically small – packet-loss rate p. "Noise" losses (losses not due to congestion) are not frequent but no longer negligible; these keep the window significantly smaller than it should be. The following table, from **RFC 3649**, is based on an RTT of 0.1 seconds and a packet size of 1500 bytes, for various throughputs. The `cwnd` values represent the bandwidth×RTT products.

TCP Throughput (Mbps)	RTTs between losses	cwnd	Packet Loss Rate P
1	5.5	8.3	0.02
10	55	83	0.0002
100	555	833	2×10^{-6}
1000	5555	8333	2×10^{-8}
10,000	55555	83333	2×10^{-10}

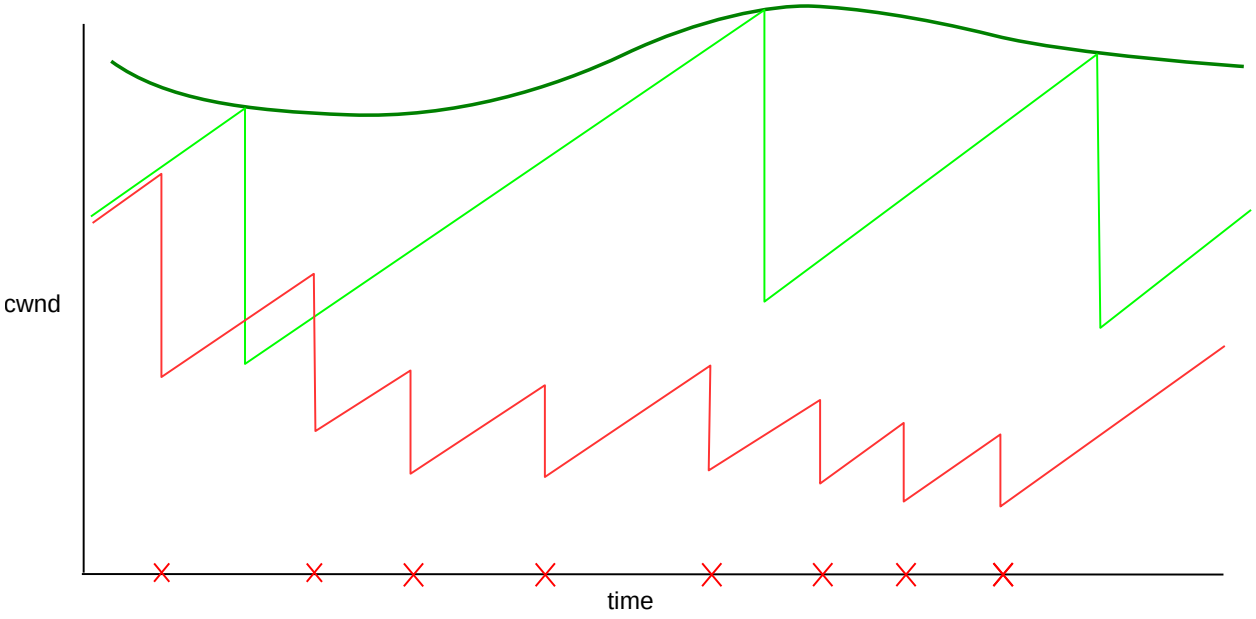
Note the very small value of the loss probability needed to support 10 Gbps; this works out to a bit error rate of less than 2×10^{-14}. For fiber optic data links, alas, a physical bit error rate of 10^{-13} is often considered acceptable; there is thus no way to support the window size of the final row above. (The use of error-correcting codes on OTN links, *6.2.3 Optical Transport Network*, can reduce the bit error rate to less than 10^{-15}.) Another source of "noise" losses are queue overflows within Ethernet switches; switches tend to have much shorter queues than routers. At 10 Gbps, a switch is forwarding one packet every microsecond; at that rate a burst does not have to last long to overrun the switch's queue.

Here is a similar table, expressing cwnd in terms of the packet loss rate:

Packet Loss Rate P	cwnd	RTTs between losses
10^{-2}	12	8
10^{-3}	38	25
10^{-4}	120	80
10^{-5}	379	252
10^{-6}	1,200	800
10^{-7}	3,795	2,530
10^{-8}	12,000	8,000
10^{-9}	37,948	25,298
10^{-10}	120,000	80,000

The above two tables indicate that large window sizes require extremely small drop rates. This is the **high-bandwidth-TCP problem**: how do we maintain a large window when a path has a large bandwidth×delay product? The primary issue is that non-congestive (noise) packet losses bring the window size down, potentially far below where it could be. A secondary issue is that, even if such random drops are not significant, the increase of cwnd to a reasonable level can be quite slow. If the network ceiling were about 2,000 packets, then the normal sawtooth return to the ceiling after a loss would take 1,000 RTTs. This is slow, but the sender would still average 75% throughput, as we saw in *19.7 TCP and Bottleneck Link Utilization*. Perhaps more seriously, if the network ceiling were to double to 4,000 packets due to decreases in competing traffic, it would take the sender an additional 2,000 RTTs to reach the point where the link was saturated.

In the following diagram, the network ceiling and the ideal TCP sawtooth are shown in green. The ideal TCP sawtooth should range between 50% and 100% of the ceiling; in the diagram, "noise" or non-congestive losses occur at the red x's, bringing down the throughput to a much lower average level.



TCP without (green) and with (red) random losses
In this diagram, red random losses occur 3-4 times as often as green congestion losses

21.7 The Lossy-Link TCP Problem

Closely related to the high-bandwidth problem is the lossy-link problem, where one link on the path has a relatively high **non-congestive-loss** rate; the classic example of such a link is Wi-Fi. If TCP is used on a path with a 1.0% loss rate, then *21.2 TCP Reno loss rate versus cwnd* indicates that the sender can expect an average `cwnd` of only about 12, no matter how high the bandwidth×delay product is.

The only difference between the lossy-link problem and the high-bandwidth problem is one of scale; the lossy-link problem involves unusually large values of p while the high-bandwidth problem involves circumstances where p is quite low *but not low enough*. For a given non-congestive loss rate p, if the bandwidth×delay product is much in excess of $1.22/\sqrt{p}$ then the sender will be unable to maintain a `cwnd` close to the network ceiling.

21.8 The Satellite-Link TCP Problem

A third TCP problem, only partially related to the previous two, is that encountered by TCP users with very long RTTs. The most dramatic example of this involves satellite Internet links (*4.4.2 Satellite Internet*). Communication each way involves routing the signal through a satellite in geosynchronous orbit; a round trip involves four up-or-down trips of ~36,000 km each and thus has a propagation delay of about 500ms. If we take the per-user bandwidth to be 1 Mbps (satellite ISPs usually provide quite limited bandwidth, though peak bandwidths can be higher), then the bandwidth×delay product is about 40 packets. This is not especially high, even when typical queuing delays of another ~500ms are included, but the fact that it takes many seconds to reach even a moderate `cwnd` is an annoyance for many applications. Most ISPs provide an "acceleration" mechanism when they can identify a TCP connection as a file download; this usually involves transferring the file over the satellite portion of the path using a proprietary protocol. However, this is not

much use to those using TCP connections that involve multiple bidirectional exchanges; *eg* those using VPN connections.

21.9 Epilog

TCP Reno's core congestion algorithm is based on algorithms in Jacobson and Karel's 1988 paper [JK88], now twenty-five years old. There are concerns both that TCP Reno uses too much bandwidth (the greediness issue) and that it does not use enough (the high-bandwidth-TCP problem).

In the next chapter we consider alternative versions of TCP that attempt to solve some of the above problems associated with TCP Reno.

21.10 Exercises

Exercises may be given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercises marked with a ♢ have solutions or hints at *34.16 Solutions for Dynamics of TCP*.

1.0. For each value α or β below, find the other value so that AIMD(α,β) is TCP-friendly.

(a). β = 1/5
(b). β = 2/9
(c). α = 1/5

Then pick the pair that has the smallest α, and draw a sawtooth diagram that is approximately proportional: α should be the slope of the linear increase, and β should be the decrease fraction at the end of each tooth.

2.0. Suppose two TCP flows compete. The flows have the same RTT. The first flow uses AIMD(α_1,β_1) and the second uses AIMD(α_2,β_2); neither flow is necessarily TCP-Reno-friendly. The two connections, however, compete fairly with one another; that is, they have the same average packet-loss rates. Show that

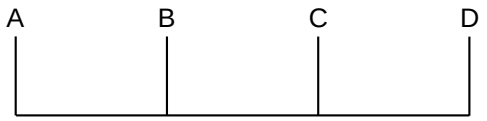
$$\alpha_1/\beta_1 = (2\text{-}\beta_2)/(2\text{-}\beta_1) \times \alpha_2/\beta_2.$$

Assume regular losses, and use the methods of *21.4 AIMD Revisited*. Hint: first, apply the argument there to show that the two flows' teeth must have the same width w and same average height. The average height is no longer 3w/2, but can still be expressed in terms of w, α and β. Use a diagram to show that, for any tooth, average_height = h$\times$(1–β/2), with h the right-edge height of the tooth. Then equate the two average heights of the h_1/β_1 and h_2/β_2 teeth. Finally, use the $\alpha_i w = \beta_i h_i$ relationships to eliminate h_1 and h_2.

3.0. Using the result of the previous exercise, show that AIMD(α_1,β) is equivalent to (in the sense of competing fairly with) AIMD(α_2,0.5), with $\alpha_2 = \alpha_1 \times (2–\beta)/3\beta$.

4.0. Suppose two 1 kB packets are sent as part of a packet-pair probe, and the minimum time measured between arrivals is 5 ms. What is the estimated bottleneck bandwidth?

5.0. Consider again the three-link parking-lot network from *21.1.1 Max-Min Fairness*:



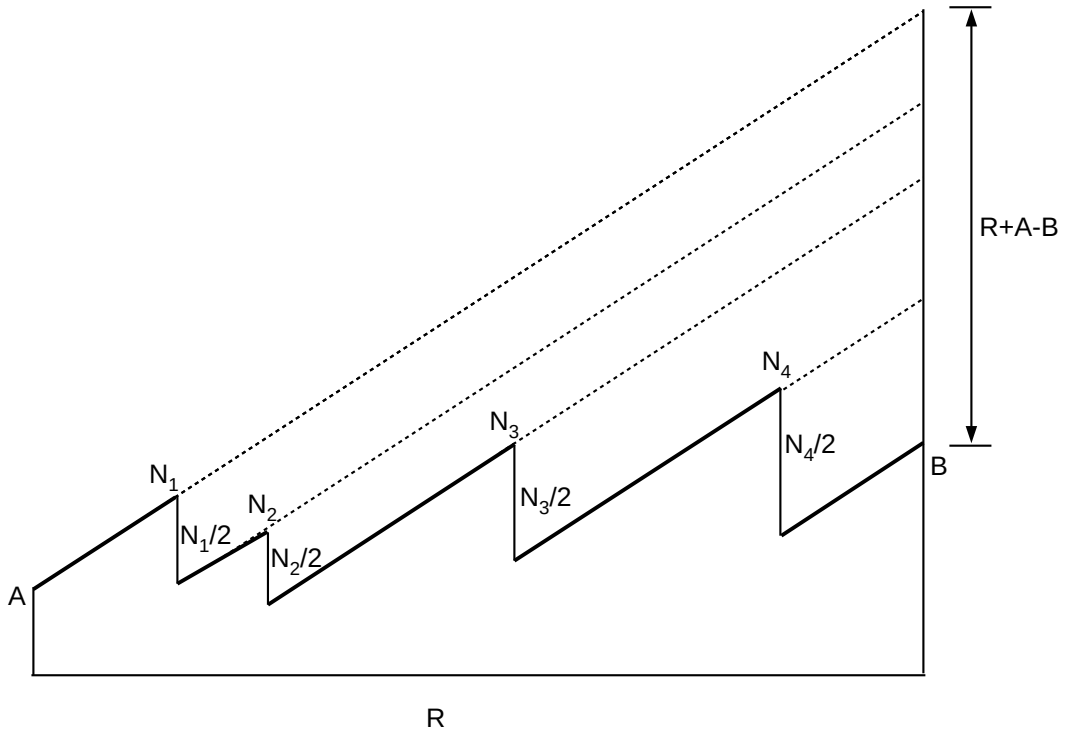
(a). Suppose we have *two* end-to-end connections, in addition to one single-link connection for each link. Find the max-min-fair allocation.

(b). Suppose we have a single end-to-end connection, and one B–C and C–D connection, but two A–B connections. Find the max-min-fair allocation.

6.0. Consider the two-link parking-lot network:

Suppose there are two A–C connections, one A–B connection and one A–C connection. Find the allocation that is proportionally fair.

7.0. Suppose we use TCP Reno to send K packets over R RTT intervals. The transmission experiences n not-necessarily-uniform loss events; the TCP `cwnd` graph thus has n sawtooth peaks of heights N_1 through N_n. At the start of the graph, `cwnd` = A, and at the end of the graph, `cwnd` = B. Show that the sum $N_1 + \ldots + N_n$ is 2(R+A-B), and in particular the average tooth height is independent of the distribution of the loss events.



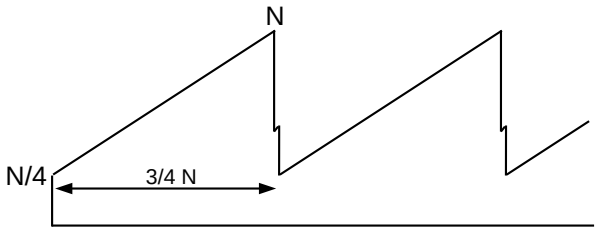
8.0. Suppose the bandwidth×delay product for a network path is 1000 packets. The only traffic on the path is from a single TCP Reno connection. For each of the following cases, find the average `cwnd` and the approximate number of packets between losses (the reciprocal of the loss rate). Your answers, collectively, should reflect the formula in *21.2 TCP Reno loss rate versus cwnd*.

(a).♢ The bottleneck queue capacity is close to zero.

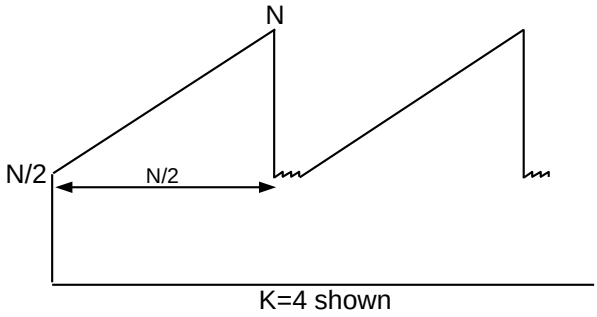
(b). The bottleneck queue capacity is 1000 packets.

(c). The bottleneck queue capacity is 3000 packets.

9.0. Suppose TCP Reno has regularly spaced sawtooth peaks of the same height, but the packet losses come in pairs, with just enough separation that both losses in a pair are counted separately. N is large enough that the spacing between the two losses is negligible. The net effect is that each large-scale tooth ranges from height N/4 to N. As in *21.2 TCP Reno loss rate versus cwnd*, $\texttt{cwnd}_{\text{mean}} = K/\sqrt{p}$ for some constant K. Find the constant. Hint: from the given information one can determine both $\texttt{cwnd}_{\text{mean}}$ and the number of packets sent in one tooth. The loss rate is p = 2/(number of packets sent in one tooth).

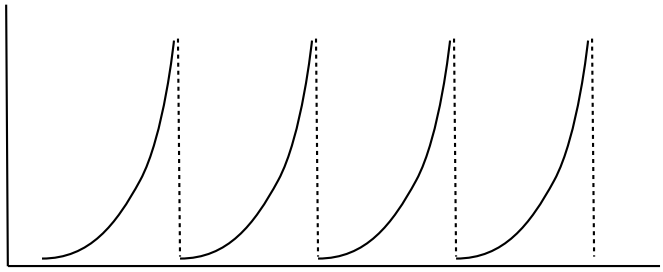


10.0. As in the previous exercise, suppose a TCP transmission has large-scale teeth of height N. Between each pair of consecutive large teeth, however, there are K-1 additional losses resulting in K-1 additional tiny teeth; N is large enough that these tiny teeth can be ignored. A non-Reno variant of TCP is used, so that between these tiny teeth `cwnd` is assumed not to be cut in half; during the course of these tiny teeth `cwnd` does not change much at all. The large-scale tooth has width N/2 and height ranging from N/2 to N, and there are K losses per large-scale tooth. Find the ratio `cwnd`/($1/\sqrt{p}$), in terms of K. When K=1 your answer should reduce to that derived in *21.2 TCP Reno loss rate versus cwnd*.



11.0. Suppose a TCP Reno tooth starts with `cwnd` = c, and contains N packets. Let w be the width of the tooth, in RTTs as usual. Show that $w = (c^2 + 2N)^{1/2} - c$. Hint: the maximum height of the tooth will be c+w, and so the average height will be c + w/2. Find an equation relating c, w and N, and solve for w using the quadratic formula.

12.0. Suppose we have a non-Reno implementation of TCP, in which the formula relating the `cwnd` c to the time t, as measured in RTTs since the most recent loss event, is $c = t^2$ (versus TCP Reno's $c = c_0 + t$). The sawtooth then looks like the following:



The number of packets sent in a tooth of width T RTTs, which is the reciprocal of the loss rate p, is now approximately $T^3/3$ (this may be accepted on faith, or shown by integrating t^2 from 0 to T, or looking up the

formula for $1^2 + 2^2 + \ldots + T^2$). The average `cwnd` is therefore $T^3/3T = T^2/3$.

Derive a formula expressing the average `cwnd` in terms of the loss rate p. Hint: the exponent for p should be –2/3, versus –1/2 in the formula in *21.2 TCP Reno loss rate versus cwnd*.

13.0. Using the TCP assumptions of exercise 12.0 above, `cwnd` is incremented by about 2t per each RTT. Show that the `cwnd` increment rule can be expressed as

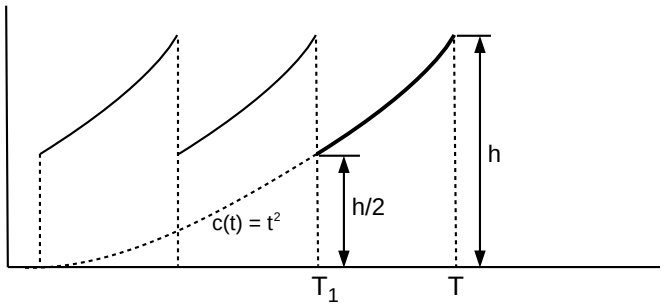
$$\texttt{cwnd} \mathrel{+}= \alpha \texttt{cwnd}^{1/2}$$

and find the value of α.

14.0. Using the same TCP assumptions as in exercise 12.0 above, show that `cwnd` is still proportional to $p^{-2/3}$, where p is the loss rate, assuming the following:

- the top boundary of each tooth follows the curve $\texttt{cwnd} = c(t) = t^2$, as before.
- each tooth has a right boundary at t=T and a left boundary at $t=T_1$, where $c(T_1) = 0.5 \times c(T)$.

(In the previous exercise we assumed, in effect, that $T_1 = 0$ and that `cwnd` dropped to 0 after each loss event; here we assume multiplicative decrease is in effect with β=1/2.) The number of packets sent in one tooth is now $k \times (T^3 - {T_1}^3)$, and the mean `cwnd` is this divided by $T–T_1$.



Note that as the teeth here become higher, they become proportionately narrower. Hint: show $T_1 = (0.5)^{0.5} \times T$, and then eliminate T_1 from the above equations.

15.0. Suppose in a TCP Reno run each packet is equally likely to be lost; the number of packets N in each tooth will therefore be distributed exponentially. That is, N = -k log(X), where X is a uniformly distributed random number in the range 0<X<1 (k, which does not really matter here, is the mean interval between losses). Write a simple program that simulates such a TCP Reno run. At the end of the simulation, output an estimate of the constant C in the formula $\texttt{cwnd}_{\text{mean}} = C/\sqrt{p}$. You should get a value of about 1.31, as in the formula in *21.2.1 Irregular teeth*.

Hint: There is no need to simulate packet transmissions; we simply create a series of teeth of random size, and maintain running totals of the number of packets sent, the number of RTT intervals needed to send them, and the number of loss events (that is, teeth). After each loss event (each tooth), we update:

- total_packets += *packets sent in this tooth*
- RTT_intervals += *RTT intervals in this tooth*
- loss_events += 1 (one tooth = one loss event)

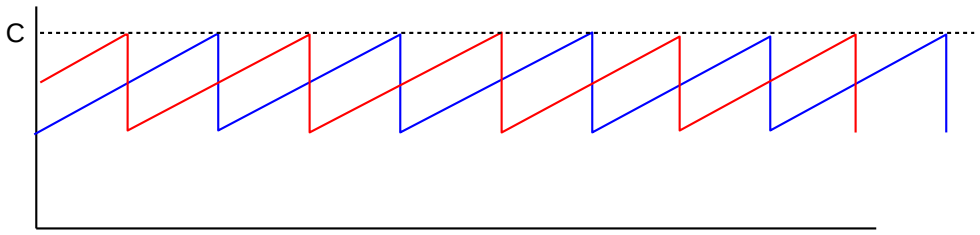
If a loss event marking the end of one tooth occurs at a specific value of `cwnd`, the next tooth begins at height c = `cwnd`/2. If N is the random value for the number of packets in this tooth, then by the previous

exercise the tooth width in RTTs is $w = (c^2 + 2N)^{1/2} - c$; the next peak (that is, loss event) therefore occurs when `cwnd` = c+w. Update the totals as above and go on to the next tooth. It should be possible to run this simulation for 1 million teeth in modest time.

16.0. Suppose two TCP connections have the same RTT and share a bottleneck link, for which there is no other competition. The size of the bottleneck queue is negligible when compared to the bandwidth $\times$ RTT_{noLoad} product. Loss events occur at regular intervals.

In Exercise 12.0 of the previous chapter, you were to show that if losses are synchronized then the two connections together will use 75% of the total bottleneck-link capacity

Now assume the two TCP connections have no losses in common, and, in fact, *alternate* losses at regular intervals as in the following diagram.



Both connections have a maximum `cwnd` of C. When Connection 1 experiences a loss, Connection 2 will have `cwnd` = 75% of C, and vice-versa.

(a). What is the combined transit capacity of the paths, in terms of C? (Because the queue size is negligible, the transit capacity is approximately the sum of the `cwnd`s at the point of loss.)

(b). Find the bottleneck-link utilization. Hint: Again because the queue size is negligible, this is approximately the ratio of the average total cwnd to the transit capacity of part (a). It should be at least 85%.

22 NEWER TCP IMPLEMENTATIONS

Since the rise of TCP Reno, several TCP alternatives to Reno have been developed; each attempts to address some perceived shortcoming of Reno. While many of them are very specific attempts to address the high-bandwidth problem we considered in *21.6 The High-Bandwidth TCP Problem*, some focus primarily or entirely on other TCP Reno foibles. One such issue is TCP Reno's "greediness" in terms of queue utilization; another is the lossy-link problem (*21.7 The Lossy-Link TCP Problem*) experienced by, say, Wi-Fi users.

Generally speaking, a TCP implementation can respond to congestion at the cliff – that is, it can respond to packet losses – or can respond to congestion at the knee – that is, it can detect the increase in RTT associated with the filling of the queue. These strategies are sometimes referred to as **loss-based** and **delay-based**, respectively; the latter term because of the rise in RTT. TCP implementers can tweak both the loss response – the multiplicative decrease of TCP Reno – and also the way TCP increases its `cwnd` in the absence of loss. There is a rich variety of options available.

The concept of monitoring the RTT to avoid congestion at the knee was first introduced in TCP Vegas (*22.6 TCP Vegas*). One striking feature of TCP Vegas is that, in the absence of competition, the queue may never fill, and thus there may not be any congestive losses. The TCP sawtooth, in other words, is not inevitable.

When losses do occur, most of the mechanisms reviewed here continue to use the TCP NewReno recovery strategy. As most of the implementations here are relatively recent, the senders can generally expect that the receiving end will support SACK TCP, which allows more rapid recovery from multiple losses.

22.1 Choosing a TCP on Linux

On Linux systems, the TCP congestion-control mechanism can be set by writing an appropriate string to `/proc/sys/net/ipv4/tcp_congestion_control` (or, equivalently, by passing the string as a parameter to the `sysctl net.ipv4.tcp_congestion_control` command). The standard options on the author's system as of 2013 are listed below (as of 2016, several are now only available if loaded explicitly, *eg* with `modprobe`). The list comes from `/proc/sys/net/ipv4/tcp_available_congestion_control`.

- highspeed
- htcp
- hybla
- illinois
- vegas
- veno
- westwood
- bic
- cubic

We review several of these below; see *22.4 A Roadmap* for an overview. TCP Cubic is currently (2013) the default Linux congestion-control implementation; TCP Bic was a precursor.

The TCP congestion-control mechanism can also be set on a per-connection basis. Non-root users can select any mechanism listed in `/proc/sys/net/ipv4/tcp_allowed_congestion_control`; entries in tcp_available_congestion_control can be copied to this by the root user.

Many TCP flavors are not available by default, but can be loaded via `modprobe`. The modules containing the TCP implementations are generally in /lib/modules/$(uname -r)/kernel/net/ipv4. Executing `ls tcp_*` in this directory yields (on the author's system in 2017) the following:

- tcp_bic.ko
- tcp_cdg.ko
- tcp_dctcp.kp
- tcp_highspeed.ko
- tcp_htcp.ko
- tcp_hybla.ko
- tcp_illinois.ko
- tcp_scalable.ko
- tcp_vegas.ko
- tcp_veno.ko
- tcp_westwood.ko
- tcp_yeah.ko

To load, *eg*, TCP Vegas, use `modprobe tcp_vegas` (without the ".ko"). This will last until the next reboot (or until the module is manually unloaded). At this point /proc/sys/net/ipv4/tcp_available_congestion_control will contain "vegas" (not tcp_vegas).

In the C language, we can select the Linux congestion control mechanism, after socket creation but before connection, by including the `setsockopt()` call below; see *28.2.2 An Actual Stack-Overflow Example* and *29.5.3 A TLS Programming Example* for complete C examples (though without this call).

```
#include <netinet/in.h>
#include <netinet/tcp.h>
...
char * cong_algorithm = "vegas";
int slen = strlen( cong_algorithm ) + 1;
int rc = setsockopt( sock, IPPROTO_TCP, TCP_CONGESTION, cong_algorithm, slen);
if (rc < 0) { /* error */ }
```

Checking the return code is essential to determine if the algorithm request succeeded.

In Python3 (and Python2) we can do this as well; the file below is also available at tcp_stalkc_cong.py. See also *30.7.3.1 sender.py*.

```
#!/usr/bin/python3
# stalk client allowing specification of the TCP congestion algorithm

from socket import *
from sys import argv

default_host = "localhost"
portnum = 5431

cong_algorithm = 'vegas'

def talk():
    global cong_algorithm
    rhost = default_host
    if len(argv) > 1:
        rhost = argv[1]
    if len(argv) > 2:
        cong_algorithm = argv[2]
    print("Looking up address of " + rhost + "...", end="")
    try:
        dest = gethostbyname(rhost)
    except gaierror as mesg:    # host not found
        errno,errstr=mesg.args
        print("\n   ", errstr);
        return;
    print("got it: " + dest)
    addr=(dest, portnum)
    s = socket()

    TCP_CONGESTION = 13   # defined in /usr/include/netinet/tcp.h
    cong = bytes(cong_algorithm, 'ascii')
    try:
       s.setsockopt(IPPROTO_TCP, TCP_CONGESTION, cong)
    except OSError as mesg:
       errno, errstr = mesg.args
       print ('congestion mechanism {} not available: {}'.format(cong_
↪algorithm, errstr))
       return

    res=s.connect_ex(addr)      # make the actual connection
    if res!=0:
        print("connect to port ", portnum, " failed")
        exit()

    while True:
        try:
            buf = input("> ")
        except:
            break;
        buf = buf + "\n"
        s.send(bytes(buf, 'ascii'))

talk()
```

As of version 3.5, Python did not define the constant `TCP_CONGESTION`; the value 13 above was found in the C include file mentioned in the comment. Fortunately, Python simply passes the parameters of `s.setsockopt()` to the underlying C call, and everything works. Supposedly `TCP_CONGESTION` is predefined in Python 3.6.

22.2 High-Bandwidth Desiderata

One goal of all TCP implementations that attempt to fix the high-bandwidth problem is to be **un**fair to TCP Reno: the whole point is to allow `cwnd` to increase more aggressively than is permitted by Reno. Beyond that, let us review what else a TCP version should do.

First is the **backwards-compatibility** constraint: any new TCP should exhibit reasonable **fairness** with TCP Reno at lower bandwidth×delay products. In particular, it should not ever have a significantly lower `cwnd` than a competing TCP Reno would get. But also it should not take bandwidth unfairly from a TCP Reno connection: the above comment about unfairness to Reno notwithstanding, the new TCP, when competing with TCP Reno, should leave the Reno connection with about the same bandwidth it would have if it were competing with another Reno connection. This is possible because at higher bandwidth×delay products TCP Reno does not efficiently use the available bandwidth; the new TCP should to the extent possible restrict itself to consuming this previously unavailable bandwidth rather than eating significantly into the bandwidth of a competing TCP Reno connection.

There is also the **self-fairness** issue: multiple connections using the new TCP should receive similar bandwidth allocations, at least with similar RTTs. For dissimilar RTTs, the bandwidth proportions should ideally be no worse than they would be under TCP Reno.

Ideally, we also want relatively **rapid convergence** to fairness; fairness is something of a hollow promise if only connections transferring more than a gigabyte will benefit from it. For TCP Reno, two connections halve the difference in their respective `cwnds` at each shared loss event; as we saw in *21.4.1 AIMD and Convergence to Fairness*, slower convergence is possible.

It is harder to hope for fairness between competing new implementations. However, at the very least, if new implementations tcp1 and tcp2 are competing, then neither should get less than TCP Reno would get.

Some new TCPs make use of careful RTT measurements, and, as we shall see below, such measurements are subject to a considerable degree of noise. Any new TCP implementation should be reasonably **robust** in the face of inaccuracies in RTT measurement; a modest or transient measurement error should not make the protocol behave badly, in either the direction of low `cwnd` or of high.

Finally, a new TCP should ideally try to avoid clusters of **multiple losses** at each loss event. Such multiple losses, for example, are a problem for TCP NewReno without SACK: as we have seen, it takes one RTT to retransmit each lost packet. Even with SACK, multiple losses complicate recovery. Yet if a new TCP increments `cwnd` by an amount N>1 after each RTT, then there is potential for the network ceiling to be exceeded by N within one RTT, making a cluster of N losses reasonably likely to occur. These losses are likely distributed among all connections, not just the new-TCP one.

All TCPs addressing the high-bandwidth issue will need a `cwnd`-increment N that is fairly large, at least some of the time, apparently conflicting with this no-multiple-losses ideal. One trick is to reduce N when packet loss appears to be imminent. TCP Illinois and TCP Cubic do have mechanisms in place to reduce multiple losses.

22.3 RTTs

The exact performance of some of the faster TCPs we consider – for that matter, the exact performance of TCP Reno – is influenced by the RTT. This may affect individual TCP performance and also competition between different TCPs. For reference, here are a few typical RTTs from Chicago to various other places:

- US West Coast: 50-100 ms
- Europe: 100-150 ms
- Southeast Asia: 100-200 ms

22.4 A Roadmap

We start with Highspeed TCP, an early and relatively simple attempt to address the high-bandwidth-TCP problem.

After that is the group TCP Vegas, FAST TCP, TCP Westwood, TCP Illinois and Compound TCP. These all involve so-called **delay-based** congestion control, in which the sender carefully monitors the RTT for the minute increases that signal queuing. TCP Vegas, which dates from 1995, is the earliest TCP here and in fact predates widespread recognition of the high-bandwidth-TCP problem. Its goal – then and now – was to prove that one could build a TCP that, in the absence of competition, could transfer arbitrarily long streams of data with no losses and with 100% bottleneck-link utilization.

The next group, consisting of TCP Veno, TCP Hybla and DCTCP, represent special-purpose TCPs. While TCP Veno may be a reasonable high-bandwidth TCP candidate, its primary goal is to improve TCP performance over lossy links such as Wi-Fi. TCP Hybla is targeted at satellite-Internet users with very long RTTs while DCTCP is for internal connections within a datacenter (which, among other things, have very short RTTs).

The last triad represents newer, non-delay-based attempts to solve the high-bandwidth-TCP problem: H-TCP, TCP Cubic and TCP BBR. TCP Cubic has become the default TCP on Linux.

22.5 Highspeed TCP

An early proposed fix for the high-bandwidth-TCP problem is HighSpeed TCP, documented in **RFC 3649** (Floyd, 2003). Highspeed TCP is sometimes called HS-TCP, but we use the longer name here to avoid confusion with the entirely unrelated H-TCP, below.

Highspeed TCP adjusts the additive-increase and multiplicative-decrease parameters α and β so that, for larger values of `cwnd`, the rate of `cwnd` increase between losses is much faster, and the `cwnd` decrease at loss events is much smaller. This allows efficient use of all the available bandwidth for large bandwidth$\times$delay products. Correspondingly, when `cwnd` is in the range where TCP Reno works well, Highspeed TCP's throughput is only modestly larger than TCP Reno's, so the two compete relatively fairly.

The threshold for Highspeed TCP diverging from TCP Reno is a loss rate less than 10^{-3}, which for TCP Reno occurs when `cwnd` = 38. Beyond that point, Highspeed TCP gradually increases α and decreases β. The overall effect is to outperform TCP Reno by a factor N = N(`cwnd`) according to the table below. This N

can also be interpreted as the "unfairness" of Highspeed TCP with respect to TCP Reno; fairness is arguably "close to" 1.0 until cwnd⩾1000, at which point TCP Reno is likely not using the full bandwidth available due to the high-bandwidth TCP problem.

cwnd	N(cwnd)
1	1.0
10	1.0
100	1.4
1,000	3.6
10,000	9.2
100,000	23.0

An algebraic expression for N(cwnd), for N⩾38, is

$$N(\texttt{cwnd}) = 0.23 \times \texttt{cwnd}^{0.4}$$

At cwnd=38 this is about 1.0; for smaller cwnd we stick with N=1.

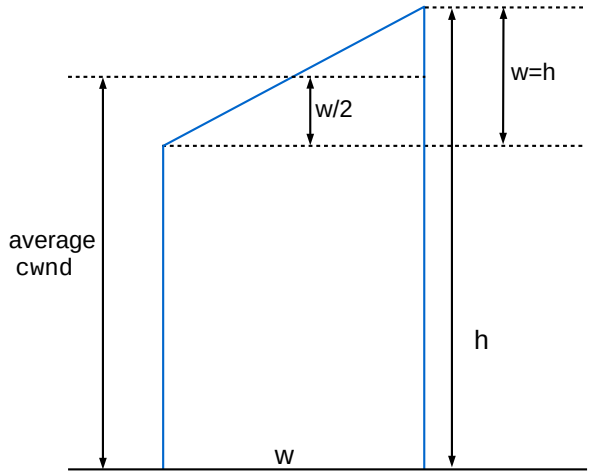
To specify the details of Highspeed TCP, we start by considering a 10 Gbps link, which was the fastest generally available at the time Highspeed TCP was developed. If the RTT is 100 ms, then the bandwidth×delay product works out to 83,000 packets. The central strategy of Highspeed TCP is to *choose* the desired loss rate for an average cwnd of 83,000 to be 1 packet in 10^7; this number was empirically determined. This is quite a bit larger than the corresponding TCP Reno loss rate of 1 packet in 5×10^9 (*21.6 The High-Bandwidth TCP Problem*); in this context, a larger congestion loss rate is better. The loss rate is the reciprocal of the tooth area; it turns out (below) that we have a great deal of latitude in choosing the tooth area by adjusting the α and β window-growth parameters. After determining α and β for cwnd = 83,000, Highspeed TCP then uses interpolation to cover cwnd values in between 38 and 83,000. (The Highspeed TCP rules do extend to larger cwnds too, but there is not necessarily an expectation that they will work well there.)

We start with the TCP Reno relationship $\texttt{cwnd} = 1.225 \times p^{-0.5}$, from *21.2 TCP Reno loss rate versus cwnd* (**RFC 3649** uses a numerator of 1.20 in this formula.) We fit the relationship $\texttt{cwnd} = k \times p^{-\alpha}$ to the above two pairs of (cwnd,p) values, $(38,10^{-3})$ and $(83000,10^{-7})$. This turns out to yield

$$\texttt{cwnd} = 0.12 \times p^{-0.835}$$

From this we can derive the TCP Reno multiplier N(cwnd) above, by using the TCP Reno relationship $\texttt{cwnd} = 1.2 \times N \times p^{-0.5}$ for N synchronized connections, eliminating p and then solving for N.

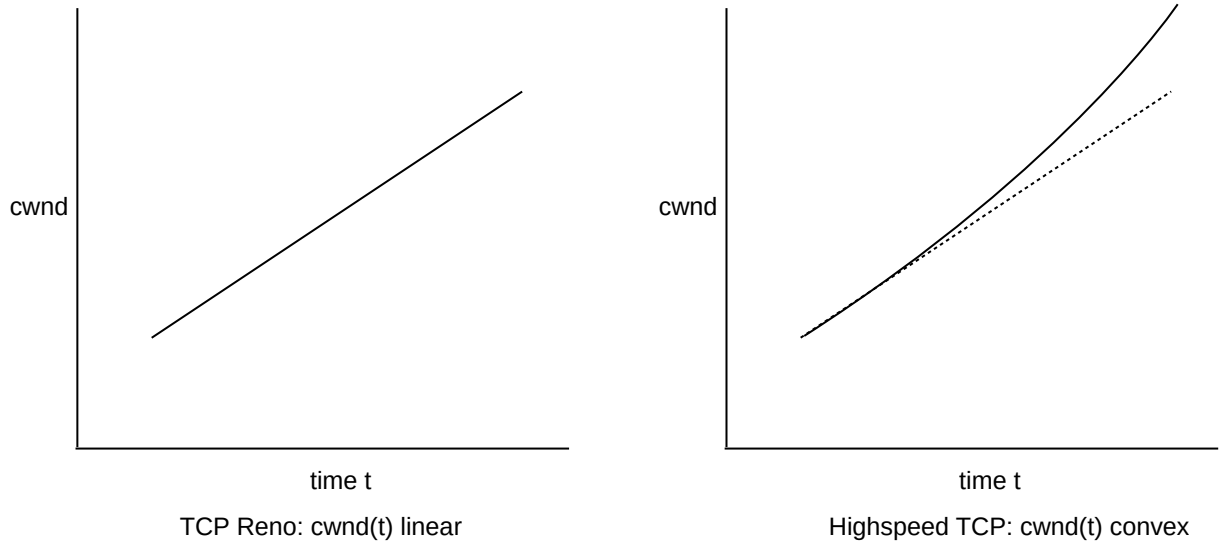
The next step is to define the additive-increase and multiplicative-decrease values $\alpha = \alpha(\texttt{cwnd})$ and $\beta = \beta(\texttt{cwnd})$, thus allowing us to build an actual implementation. While α and β are allowed to vary with cwnd, we will assume they do so only slowly, so that for any given steady-state connection the α values are relatively constant (the β value is that at the maximum cwnd). This gives us a standard AIMD tooth:



When `cwnd` is 83,000 we want the loss rate to be 10^{-7}, meaning that the area of the tooth, w×`cwnd` = w×h×(1-β/2), should be 10^7. From this we get w = 10^7/83,000 = 120.5 RTTs. We also have, very generally, αw = βh, and combining this with `cwnd` = h×(1-β/2), we get α = βh/w = `cwnd`×(2β/(1-β/2))/w $\simeq$ 1378×β/(1-β/2). **RFC 3649** suggests β=0.1 at this `cwnd`, making α = 73. The value of β for values of `cwnd` between 38 and 83,000 is determined by logarithmic interpolation between 0.5 and 0.1; the corresponding value of α(`cwnd`) is then set by the formula.

The 1-in-10^7 loss rate – corresponding to a bit error rate of about one in 1.2×10^{11} – is large enough that it is at least two orders of magnitude higher than the rate of noise-induced non-congestive packet losses. On the other hand, it is small enough that the Highspeed TCP derived from it competes reasonably fairly with TCP Reno, at least with bandwidth×delay products small enough that TCP Reno alone performs reasonably well.

It may be helpful to view Highspeed TCP in terms of the `cwnd` graph between losses. For ordinary TCP, the graph increases linearly. For Highspeed TCP, the graph is slightly **convex** (lying above its tangent). This means that there is a modest increase in the *rate* of `cwnd` increase, as time goes on (up to the point of packet loss).



This might be an appropriate time to point out that in TCP Reno, the `cwnd`-versus-**time** graph between losses is actually slightly **concave** (lying below its tangent). We do get a strictly linear graph if we plot `cwnd` as a function of the count of elapsed RTTs, but the RTTs are themselves slowly increasing as a function of time once the queue starts filling up. At that point, the `cwnd`-versus-time graph bends slightly down. If the bottleneck queue capacity matches the total path transit capacity, the RTTs for a full queue are about double the RTTs for an empty queue.

In general, when Highspeed-TCP competes with a new TCP Reno flow it is N times as aggressive, and grabs N times the bandwidth, where N = N(`cwnd`) is as above. For `cwnd` = 83,000, the formula above yields N = 21. This may be surprising, as for this value of `cwnd` Highspeed TCP is AIMD(73,0.1), which is equivalent to AIMD(459,0.5) (either via the formula above or by *21.10 Exercises*, exercise 2.0). We might naively suppose that AIMD(459,0.5) would out-compete TCP Reno – AIMD(1,0.5) – by a factor of 459, by the reasoning of *20.3.1 Example 2: Faster additive increase*. But this is true only if losses are synchronized, which, for such lopsided differences in α, is manifestly not the case. Because Highspeed TCP uses the lion's share of the queue, it encounters the lion's share of loss events, and TCP Reno is able to do much better than the α values alone would suggest.

Finally, with a little math we can compare Highspeed TCP with an AIMD-type flavor of TCP with an additive-increase rule (per RTT) of the form

$$\texttt{cwnd} \mathrel{+}= \alpha \times \texttt{cwnd}^{\texttt{k}}$$

For TCP Reno, k=0, and in the example of exercises 12.0 and 13.0 of *21.10 Exercises* we have k=1/2. For compatibility with Highspeed TCP, it turns out what we need is k=0.8. We will return to this in *22.10 Compound TCP*, which intentionally mimics the behavior of Highspeed TCP when queue utilization is low.

22.6 TCP Vegas

TCP Vegas, introduced in [BP95], is the only new TCP version we consider here that dates from the previous century. The goal was not directly to address the high-bandwidth problem, but rather to improve TCP throughput generally; indeed, in 1995 the high-bandwidth problem had not yet surfaced as a practical con-

cern. The ambitious goal of TCP Vegas is essentially to eliminate congestive losses, and to try to keep the bottleneck link 100% utilized at all times. Recall from *19.7 TCP and Bottleneck Link Utilization* that, with a large queue, the average bottleneck-link utilization for TCP Reno can be as low as 75%.

TCP Vegas achieves this improvement by, like DECbit, recognizing TCP congestion at the *knee*, that is, at the point where the bottleneck link has become saturated and further `cwnd` increases simply result in RTT increases. A TCP Vegas sender alone or in competition only with other TCP Vegas connections will seldom if ever approach the "cliff" where packet losses occur.

To accomplish this, no special router cooperation – or even receiver cooperation – is necessary. Instead, the sender uses careful monitoring of the RTT to keep track of the number of "extra packets" (*ie* packets sitting in queues) it has injected into the network. In the absence of competition, the RTT will remain constant, equal to $\text{RTT}_{\text{noLoad}}$, until `cwnd` has increased to the point when the bottleneck link has become saturated and the queue begins to fill (*8.3.2 RTT Calculations*). By monitoring the bandwidth as well, a TCP sender can even determine the actual number of packets in the bottleneck queue, as bandwidth $\times$ (RTT – $\text{RTT}_{\text{noLoad}}$). TCP Vegas uses this information to attempt to maintain at all times a small but positive number of packets in the bottleneck queue.

This TCP Vegas strategy is now often referred to as **delay-based congestion control**, as opposed to TCP Reno's **loss-based** congestion control. TCP Reno's periodic losses followed by the halving of `cwnd` is what leads to the "TCP sawtooth"; TCP Vegas, however, has no sawtooth.

A TCP sender can readily measure its throughput. The simplest measurement is `cwnd`/RTT as in *8.3.2 RTT Calculations*; this amounts to averaging throughput over an entire RTT. Let us denote this bandwidth estimate by BWE; for the time being we will accept BWE as accurate, though see *22.8.1 ACK Compression and Westwood+* below. TCP Vegas estimates $\text{RTT}_{\text{noLoad}}$ by the minimum RTT (RTT_{min}) encountered during the connection. The "ideal" `cwnd` that just saturates the bottleneck link is $\text{BWE}\times\text{RTT}_{\text{noLoad}}$. Note that BWE will be much more volatile than RTT_{min}; the latter will typically reach its final value early in the connection, while BWE will fluctuate up and down with congestion (which will also act on RTT, but by increasing it).

As in *8.3.2 RTT Calculations*, any TCP sender can estimate queue utilization as

$$\text{queue\_use} = \texttt{cwnd} - \text{BWE}\times\text{RTT}_{\text{noLoad}} = \texttt{cwnd} \times (1 - \text{RTT}_{\text{noLoad}}/\text{RTT}_{\text{actual}})$$

TCP Vegas then adjusts `cwnd` regularly to maintain the following:

$$\alpha \leqslant \text{queue\_use} \leqslant \beta$$

which is the same as

$$\text{BWE}\times\text{RTT}_{\text{noLoad}} + \alpha \leqslant \texttt{cwnd} \leqslant \text{BWE}\times\text{RTT}_{\text{noLoad}} + \beta$$

Typically α = 2-3 packets and β = 4-6 packets. We increment `cwnd` by 1 if `cwnd` falls below the lower limit (*eg* if BWE has increased). Similarly, we decrement `cwnd` by 1 if BWE drops and `cwnd` exceeds $\text{BWE}\times\text{RTT}_{\text{noLoad}} + \beta$. These adjustments are conceptually done once per RTT. Typically a TCP Vegas sender would also set `cwnd` = `cwnd`/2 if a packet were actually lost, though this does not necessarily happen nearly as often as with TCP Reno.

TCP Vegas achieves its goal quite well. If one monitors the number of packets in queues, through real measurement or in simulation, the number does indeed stay between α and β. In the absence of competition from TCP Reno, a single TCP Vegas connection will *never* experience congestive packet loss. This is a remarkable achievement.

The use of returning ACKs to determine BWE is subject to errors due to "ACK compression", *22.8.1 ACK Compression and Westwood+*. This is generally not a major problem with TCP Vegas, however.

22.6.1 TCP Vegas versus TCP Reno

Despite its striking ability to avoid congestive losses in the absence of competition, TCP Vegas encounters a potentially serious fairness problem when competing with TCP Reno, at least for the case when queue capacity exceeds or is close to the transit capacity (*19.7 TCP and Bottleneck Link Utilization*). TCP Vegas will try to minimize its queue use, while TCP Reno happily fills the queue. And whoever has more packets in the queue has a proportionally greater share of bandwidth.

To make this precise, suppose we have two TCP connections sharing a bottleneck router R, the first using TCP Vegas and the second using TCP Reno. Suppose further that both connections have a path transit capacity of 10 packets, and R's queue capacity is 40 packets. If α=3 and β=5, TCP Vegas might keep an average of four packets in the queue. Unfortunately, TCP Reno then gobbles up most of the rest of the queue space, as follows. There are 40-4 = 36 spaces left in the queue after TCP Vegas takes its quota, and 10 in the TCP Reno connection's path, for a total of 46. This represents the TCP Reno connection's network ceiling, and is the point at which TCP Reno halves `cwnd`; therefore `cwnd` will vary from 23 to 46 with an average of about 34. Of these 34 packets, if 10 are in transit then 24 are in R's queue. If on average R has 24 packets from the Reno connection and 4 from the Vegas connection, then the bandwidth available to these connections will also be in this same 6:1 proportion. The TCP Vegas connection will get 1/7 the bandwidth, because it occupies 1/7 the queue, and the TCP Reno connection will take the other 6/7.

To put it another way, TCP Vegas is potentially too "civil" to compete with TCP Reno.

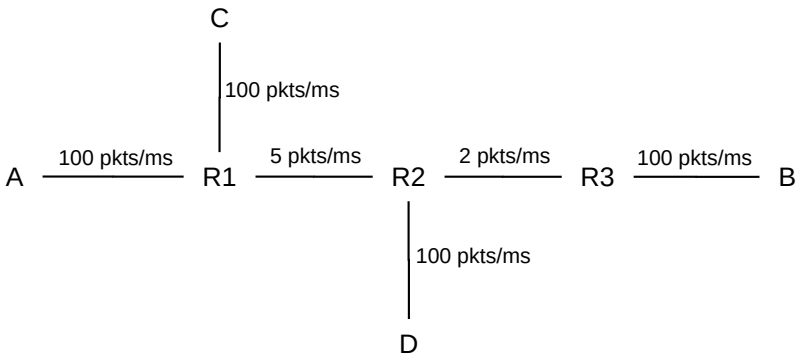
Even worse, Reno's aggressive queue filling will eventually force the TCP Vegas `cwnd` to decrease; see Exercise 4.0 below.

This Vegas-Reno fairness problem is most significant when the queue size is an appreciable fraction of the path transit capacity. During periods when the queue is empty, TCPs Vegas and Reno increase `cwnd` at the same rate, so when the queue size is small compared to the path capacity, TCP Vegas and TCP Reno are much closer to being fair.

In *31.5 TCP Reno versus TCP Vegas* we compare TCP Vegas with TCP Reno in simulation. With a transit capacity of 220 packets and a queue capacity of 10 packets, TCPs Vegas and Reno receive almost exactly the same bandwidth.

TCP Reno's advantage here assumes a router with a single FIFO queue. That advantage can disappear if a different queuing discipline is in effect. For example, if the bottleneck router used fair queuing (to be introduced in *23.5 Fair Queuing*) on a per-connection basis, then the TCP Reno connection's queue greediness would not be of any benefit, and both connections would get similar shares of bandwidth with the TCP Vegas connection experiencing lower delay. See *23.6.1 Fair Queuing and Bufferbloat*.

Let us next consider how TCP Vegas behaves when there is an increase in RTT due to the kind of cross traffic shown in *20.2.4 Example 4: cross traffic and RTT variation* and again in the diagram below. Let A–B be the TCP Vegas connection and assume that its queue-size target is 4 packets (*eg* α=3, β=5). We will also assume that the $\text{RTT}_{\text{noLoad}}$ for the A–B path is about 5ms and the RTT for the C–D path is also low. As before, the link labels represent bandwidths in packets/ms, meaning that the round-trip A–B transit capacity is 10 packets.



Initially, in the absence of C–D traffic, the A–B connection will send at a rate of 2 packets/ms (the R2–R3 bottleneck), and maintain a queue of four packets at R2. Because the RTT transit capacity is 10 packets, this will be achieved with a window size of 10+4 = 14.

Now let the C–D traffic start up, with a winsize so as to keep about four times as many packets in R1's queue as A, once the new steady-state is reached. If all four of the A–B connection's "queue" packets end up now at R1 rather than R2, then C would need to contribute at least 16 packets. These 16 packets will add a delay of about 16/5 $\simeq$ 3ms; the A–B path will see a more-or-less-fixed 3ms increase in RTT. A will also see a decrease in bandwidth due to competition; with C consuming 80% of R1's queue, A's share wll fall to 20% and thus its bandwidth will fall to 20% of the R1–R2 link bandwidth, that is, 1 packet/ms. Denote this new value by BWE_{new}. TCP Vegas will attempt to decrease `cwnd` so that

$$\texttt{cwnd} \simeq \text{BWE}_{\text{new}} \times \text{RTT}_{\text{noLoad}} + 4$$

A's estimate of $\text{RTT}_{\text{noLoad}}$, as RTT_{min}, will not change; the RTT has gotten larger, not smaller. So we have $\text{BWE}_{\text{new}} \times \text{RTT}_{\text{noLoad}} \simeq 1$ packet/ms $\times$ 5 ms = 5 packets; adding the 4 reserved for the queue, the new value of `cwnd` is now about 9, down from 14.

On the one hand, this new value of `cwnd` does represent 5 packets now in transit, plus 4 in R1's queue; this is indeed the correct response. But note that this division into transit and queue packets is an average. The actual physical A–B round-trip transit capacity remains about 10 packets, meaning that if the new packets were all appropriately spaced then *none* of them might be in any queue.

22.7 FAST TCP

FAST TCP is closely related to TCP Vegas; the idea is to keep the fixed-queue-utilization feature of TCP Vegas to the extent possible, but to provide overall improved performance, in particular in the face of competition with TCP Reno. Details can be found in [JWL04] and [WJLH06]. FAST TCP is patented; see patent 7,974,195.

As with TCP Vegas, the sender estimates $\text{RTT}_{\text{noLoad}}$ as RTT_{min}. At regular short **fixed** intervals (*eg* 20ms) `cwnd` is updated via the following weighted average:

$$\texttt{cwnd}_{\text{new}} = (1\text{-}\gamma) \times \texttt{cwnd} + \gamma \times ((\text{RTT}_{\text{noLoad}}/\text{RTT}) \times \texttt{cwnd} + \alpha)$$

where γ is a constant between 0 and 1 determining how "volatile" the `cwnd` update is ($\gamma \simeq 1$ is the most volatile) and α is a fixed constant, which, as we will verify shortly, represents the number of packets the

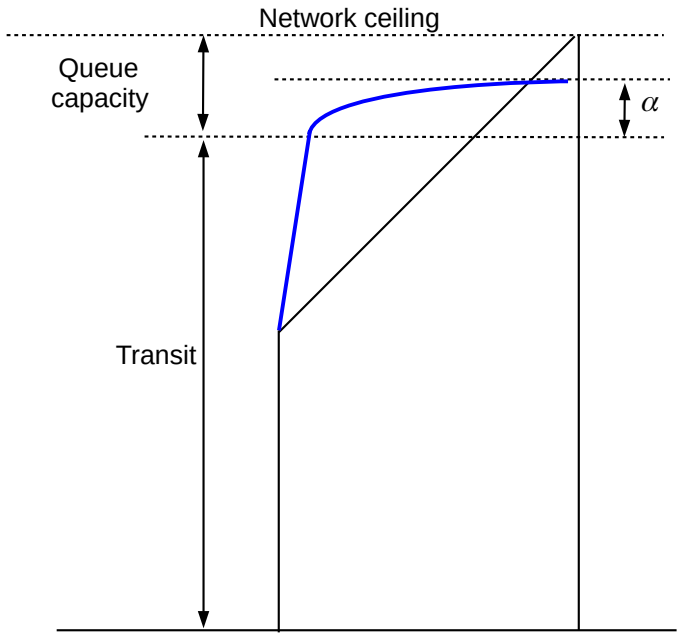
sender tries to keep in the bottleneck queue, as in TCP Vegas. Note that the `cwnd` update frequency is *not* tied to the RTT.

If RTT is constant for multiple consecutive update intervals, and is larger than $\text{RTT}_{\text{noLoad}}$, the above will converge to a constant `cwnd`, in which case we can solve for it. Convergence implies $\texttt{cwnd}_{\text{new}} = \texttt{cwnd} = ((\text{RTT}_{\text{noLoad}}/\text{RTT}) \times \texttt{cwnd} + \alpha)$, and from there we get $\texttt{cwnd} \times (\text{RTT}–\text{RTT}_{\text{noLoad}})/\text{RTT} = \alpha$. As we saw in *8.3.2 RTT Calculations*, `cwnd`/RTT is the throughput, and so α = throughput $\times$ $(\text{RTT}–\text{RTT}_{\text{noLoad}})$ is then the number of packets in the queue. In other words, FAST TCP, when it reaches a steady state, leaves α packets in the queue. As long as this is the case, the queue will not overflow (assuming α is less than the queue capacity).

Whenever the queue is not full, though, we have $\text{RTT} = \text{RTT}_{\text{noLoad}}$, in which case FAST TCP's `cwnd`-update strategy reduces to $\texttt{cwnd}_{\text{new}} = \texttt{cwnd} + \gamma \times \alpha$. For γ=0.5 and α=10, this increments `cwnd` by 5. Furthermore, FAST TCP performs this increment at a specific rate independent of the RTT, *eg* every 20ms; for long-haul links this is much less than the RTT. FAST TCP will, in other words, increase `cwnd` very aggressively until the point when queuing delays occur and RTT begins to increase.

When FAST TCP is competing with TCP Reno, it does not directly address the queue-utilization competition problem experienced by TCP Vegas. FAST TCP will try to limit its queue utilization to α; TCP Reno, however, will continue to increase its `cwnd` until the queue is full. Once the queue begins to fill, TCP Reno will pull ahead of FAST TCP just as it did with TCP Vegas. However, FAST TCP does not *reduce* its `cwnd` in the face of TCP Reno competition as quickly as TCP Vegas.

Additionally, FAST TCP can often offset this Reno-competition problem in other ways as well. First, the value of α can be increased from the value of around 4 packets originally proposed for TCP Vegas; in [TWHL05] the value α=30 is suggested. Second, for high bandwidth$\times$delay products, the queue-filling phase of a TCP Reno sawtooth (see *19.7 TCP and Bottleneck Link Utilization*) becomes relatively smaller. In the earlier link-unsaturated phase of each sawtooth, TCP Reno increases `cwnd` by 1 each RTT. As noted above, however, FAST TCP is allowed to increase `cwnd` much more rapidly in this earlier phase, and so FAST TCP can get substantially ahead of TCP Reno. It may fall back somewhat during the queue-filling phase, but overall the FAST and Reno flows may compete reasonably fairly.



FAST TCP cwnd curve (blue)
superimposed on TCP Reno sawtooth

The diagram above illustrates a FAST TCP graph of `cwnd` versus time, in blue; it is superimposed over one sawtooth of TCP Reno with the same network ceiling. Note that `cwnd` rises rapidly when it is below the path transit capacity, and then levels off sharply.

22.8 TCP Westwood

TCP Westwood represents an attempt to use the RTT-monitoring strategies of TCP Vegas to address the high-bandwidth problem; recall that the issue there is to distinguish between congestive and non-congestive losses. TCP Westwood can also be viewed as a refinement of TCP Reno's `cwnd`=`cwnd`/2 strategy, which is a greater drop than necessary if the queue capacity at the bottleneck router is less than the transit capacity. It remains a form of loss-based congestion control.

As in TCP Vegas, the sender keeps a continuous estimate of bandwidth, BWE, and estimates $\text{RTT}_{\text{noLoad}}$ by RTT_{min}. The minimum window size to keep the bottleneck link busy is, again as in TCP Vegas, BWE $\times$ $\text{RTT}_{\text{noLoad}}$. In TCP Vegas, BWE was calculated as `cwnd`/RTT; we will continue to use this for the time being but in fact TCP Westwood has used a wide variety of other algorithms, some of which are discussed in the following subsection, to infer the available average bandwidth from the returning ACKs.

The core TCP Westwood innovation is to, on loss, reduce `cwnd` as follows:

`cwnd` = max(`cwnd`/2, BWE$\times\text{RTT}_{\text{noLoad}}$) if `cwnd` > BWE$\times\text{RTT}_{\text{noLoad}}$
no change, if `cwnd` $\leqslant$ BWE$\times\text{RTT}_{\text{noLoad}}$

The product BWE$\times\text{RTT}_{\text{noLoad}}$ represents what the sender believes is its current share of the "transit capacity" of the path. This product represents how many packets can be in transit (rather than in queues) at the current bandwidth BWE. The $\text{RTT}_{\text{noLoad}}$ estimate as RTT_{min} is relatively constant, but BWE may be

markedly reduced in the presence of competing traffic. A TCP Westwood sender never drops `cwnd` below what it believes to be the current transit capacity for the path.

Consider again the classic TCP Reno sawtooth behavior:

- `cwnd` alternates between $\texttt{cwnd}_{min}$ and $\texttt{cwnd}_{max} = 2 \times \texttt{cwnd}_{min}$.
- $\texttt{cwnd}_{max} \simeq$ transit_capacity + queue_capacity (or at least the sender's share of these)

As we saw in *19.7 TCP and Bottleneck Link Utilization*, if transit_capacity < $\texttt{cwnd}_{min}$, then Reno does a reasonably good job keeping the bottleneck link saturated. However, if transit_capacity > $\texttt{cwnd}_{min}$, then when Reno drops to $\texttt{cwnd}_{min}$, the bottleneck link is not saturated until `cwnd` climbs to transit_capacity. For high-speed networks, this latter case is the more likely one.

Westwood, on the other hand, would in that situation reduce `cwnd` only to transit_capacity, a smaller reduction. Thus TCP Westwood potentially better handles a wide range of router queue capacities. For bottleneck routers where the queue capacity is small compared to the transit capacity, TCP Westwood would in theory have a higher, finer-pitched sawtooth than TCP Reno: the teeth would oscillate between the network ceiling (= queue+transit) and the transit_capacity, versus Reno's oscillation between the network ceiling and half the ceiling.

In the event of a non-congestive (noise-related) packet loss, if it happens that `cwnd` is less than transit_capacity then TCP Westwood does not reduce the window size at all. That is, non-congestive losses with `cwnd` < transit_capacity have no effect. When `cwnd` > transit_capacity, losses reduce `cwnd` only to transit_capacity, and thus the link stays saturated. This can be useful in lossy wireless environments; see [MCGSW01].

In the large-`cwnd`, high-bandwidth case, non-congestive packet losses can easily lower the TCP Reno `cwnd` to well below what is necessary to keep the bottleneck link saturated. In TCP Westwood, on the other hand, the average `cwnd` may be lower than it would be without the non-congestive losses, but it will be high enough to keep the bottleneck link saturated.

TCP Westwood uses $\text{BWE} \times \text{RTT}_{\text{noLoad}}$ as a *floor* for reducing `cwnd`. TCP Vegas shoots to have the actual `cwnd` be just a few packets *above* this.

TCP Westwood is not any more aggressive than TCP Reno at increasing `cwnd` in no-loss situations. So while it handles the non-congestive-loss part of the high-bandwidth TCP problem very well, it does not particularly improve the ability of a sender to take advantage of a sudden large rise in the network ceiling.

TCP Westwood is also potentially very effective at addressing the lossy-link problem, as most non-congestive losses would result in no change to `cwnd`.

22.8.1 ACK Compression and Westwood+

So far, we have been assuming that ACKs never encounter queuing delays. They in fact will not, *if* they are traveling in the reverse direction from all *data* packets. But while this scenario covers any single-sender model and also systems of two or more competing senders, real networks have more complicated traffic patterns, and returning ACKs from an A⟶B data flow can indeed experience queuing delays if there is third-party traffic along some link in the B⟶A path.

Delay in the delivery of ACKs, leading to clustering of their arrival, is known as **ACK compression**; see [ZSC91] and [JM92] for examples. ACK compression causes two problems. First, arriving clusters of ACKs

trigger corresponding bursts of data transmissions in sliding-windows senders; the end result is an uneven data-transmission rate. Normally the bottleneck-router queue can absorb an occasional burst; however, if the queue is nearly full such bursts can lead to premature or otherwise unexpected losses.

The second problem with late-arriving ACKs is that they can lead to inaccurate or fluctuating measurements of bandwidth, upon which both TCP Vegas and TCP Westwood depend. For example, if bandwidth is estimated as `cwnd`/RTT, late-arriving ACKs can lead to inaccurate calculation of RTT. The original TCP Westwood strategy was to estimate bandwidth from the spacing between consecutive ACKs, much as is done with the packet-pairs technique (*20.2.6 Packet Pairs*) but smoothed with a suitable running average. This strategy turned out to be particularly vulnerable to ACK-compression errors.

For TCP Vegas, ACK compression means that occasionally the sender's `cwnd` may fail to be decremented by 1; this does not appear to be a significant impact, perhaps because `cwnd` is changed by at most ± 1 each RTT. For Westwood, however, if ACK compression happens to be occurring at the instant of a packet loss, then a resultant transient overestimation of BWE may mean that the new post-loss `cwnd` is too large; at a point when `cwnd` was supposed to fall to the transit capacity, it may fail to do so. This means that the sender has essentially taken a congestion loss to be non-congestive, and ignored it. The influence of this ignored loss will persist – through the much-too-high value of `cwnd` – until the following loss event.

To fix these problems, TCP Westwood has been amended to **Westwood+**, by increasing the time interval over which bandwidth measurements are made and by inclusion of an averaging mechanism in the calculation of BWE. Too much smoothing, however, will lead to an inaccurate BWE just as surely as too little.

Suitable smoothing mechanisms are given in [FGMPC02] and [GM03]; the latter paper in particular examines several smoothing algorithms in terms of their resistance to *aliasing effects*: the tendency for intermittent measurement of a periodic signal (the returning ACKs) to lead to much greater inaccuracy than might initially be expected. One smoothing filter suggested by [GM03] is to measure BWE only over entire RTTs, and then to keep a cumulative running average as follows, where BWM_k is the measured bandwidth over the kth RTT:

$$\text{BWE}_k = \alpha \times \text{BWE}_{k-1} + (1-\alpha) \times \text{BWM}_k$$

A suggested value of α is 0.9. For Westwood+ simulations, see [GM04].

22.9 TCP Illinois

The general idea behind TCP Illinois, described in [LBS06], is to use the usual AIMD(α,β) strategy but to have $\alpha = \alpha(\text{RTT})$ be a decreasing function of the current RTT, rather than a constant. When the queue is empty and RTT is equal to $\text{RTT}_{\text{noLoad}}$, then α will be large, and `cwnd` will increase rapidly. Once RTT starts to increase, however, α will decrease rapidly, and the `cwnd` growth will level off. This leads to the same kind of concave `cwnd` graph as we saw above in FAST TCP; a consequence of this is that for most of the time between consecutive loss events `cwnd` is large enough to keep the bottleneck link close to saturated, and so to keep throughput high.

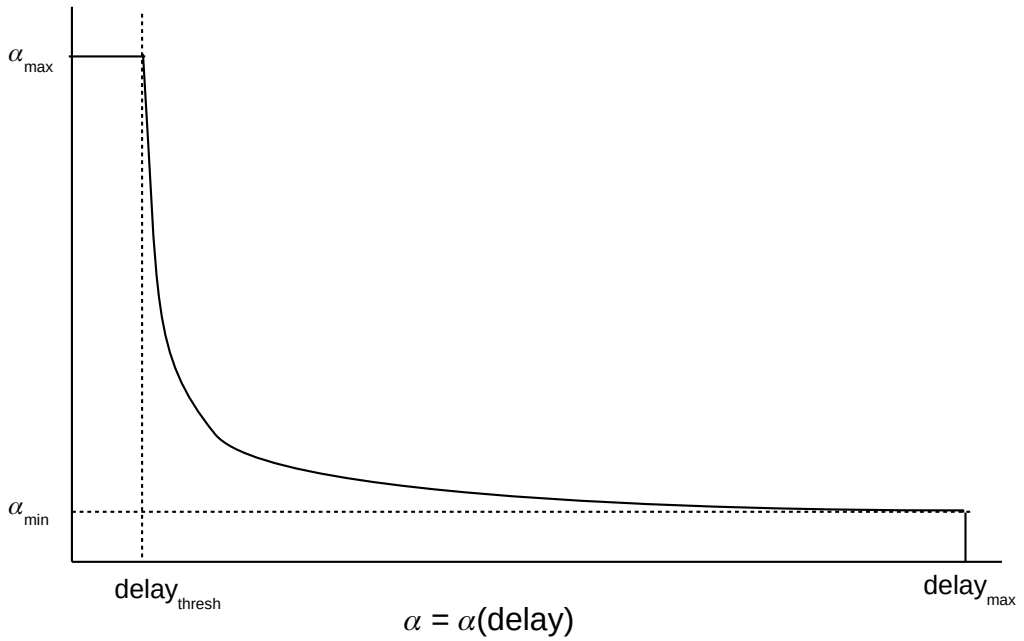
The actual $\alpha()$ function is not of RTT, but rather of `delay`, defined to be RTT – $\text{RTT}_{\text{noLoad}}$. As with TCP Vegas, $\text{RTT}_{\text{noLoad}}$ is estimated by RTT_{min}. As a connection progresses, the sender maintains continually updated values not only for RTT_{min} but also for RTT_{max}. The sender then sets $\text{delay}_{\text{max}}$ to be $\text{RTT}_{\text{max}} - \text{RTT}_{\text{min}}$.

We are now ready to define $\alpha(\text{delay})$. We first specify the highest value of α, α_{max}, and the lowest, α_{min}. In

[LBS06] these are 10.0 and 0.1 respectively; in the Linux 3.5 kernel they are 10.0 and 0.3. We also define $\text{delay}_{\text{thresh}}$ to be $0.01 \times \text{delay}_{\text{max}}$ (the 0.01 is another tunable parameter). We then define α(delay) as follows

$\alpha(\text{delay}) = \alpha_{\text{max}}$ if $\text{delay} \leqslant \text{delay}_{\text{thresh}}$
$\alpha(\text{delay}) = k_1/(\text{delay}+k_2)$ if $\text{delay}_{\text{thresh}} \leqslant \text{delay} \leqslant \text{delay}_{\text{max}}$

where k_1 and k_2 are chosen so that, for the lower formula, $\alpha(\text{delay}_{\text{thresh}}) = \alpha_{\text{max}}$ and $\alpha(\text{delay}_{\text{max}}) = \alpha_{\text{min}}$. In case there is a sudden spike in delay, $\text{delay}_{\text{max}}$ is updated before the above is evaluated, so we always have $\text{delay} \leqslant \text{delay}_{\text{max}}$. Here is a graph:



Whenever RTT = $\text{RTT}_{\text{noLoad}}$, delay=0 and so $\alpha(\text{delay}) = \alpha_{\text{max}}$. However, as soon as queuing delay just barely starts to begin, we will have delay > $\text{delay}_{\text{thresh}}$ and so α(delay) begins to fall – rather precipitously – to α_{min}. The value of α(delay) is always positive, though, so `cwnd` will continue to increase (unlike TCP Vegas) until a congestive loss eventually occurs. However, at that point the change in `cwnd` is very small, which minimizes the probability that multiple packets will be lost.

Note that, as with FAST TCP, the increase in delay is used to trigger the reduction in α.

TCP Illinois also supports having β be a decreasing function of delay, so that β(small_delay) might be 0.2 while β(larger_delay) might match TCP Reno's 0.5. However, the authors of [LBS06] explain that "the adaptation of β as a function of average queuing delay is only relevant in networks where there are non-congestion-related losses, such as wireless networks or extremely high speed networks".

22.10 Compound TCP

Compound TCP, or **CTCP**, is Microsoft's entry into the advanced-TCP field, although it is now available for Linux as well; see [TSZS06]. The idea behind Compound TCP is to add a delay-based component to TCP Reno. To this end, CTCP supplements TCP Reno's `cwnd` with a delay-based contribution to the window size known as `dwnd`; the total window size is then

winsize = `cwnd` + `dwnd`

(As usual, winsize is also not allowed to exceed the receiver's advertised window size.) The per-RTT increment of `cwnd` is now 1/winsize (though note that `dwnd` has a separate per-RTT increment).

As in TCP Vegas, CTCP maintains RTT_{min} as a stand-in for RTT_{noLoad}, and also maintains a bandwidth estimate BWE = winsize/RTT_{actual}. These allow estimation of the current number of packets in the queue, denoted `diff` in [TSZS06], as `diff` = `cwnd` $\times$ (1 – $RTT_{noLoad}/RTT_{actual}$).

When `diff` is less than γ packets, where the parameter γ is configurable but γ=30 is a good starting point, CTCP increases winsize (per RTT) according to the rule

winsize += $\alpha \times$winsize^k

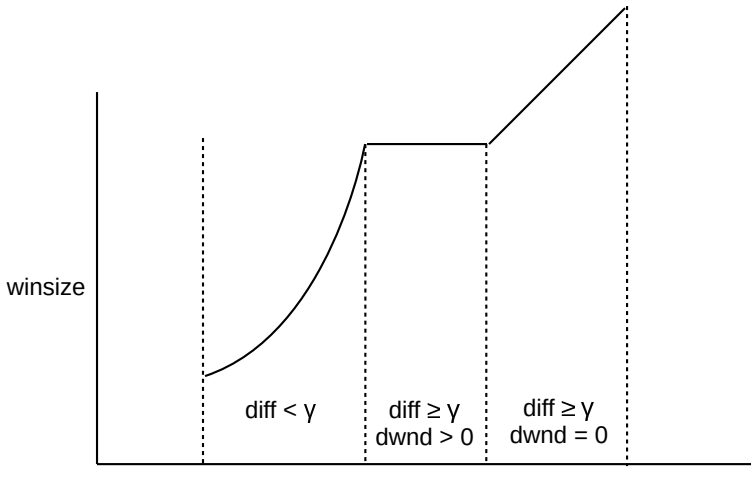
where the exponent k is chosen to be 0.8. (In [TSZS06] this increase is achieved by having `cwnd` be incremented by 1, and `dwnd` by $\alpha \times$winsize^k – 1.) This amounts to a fairly aggressive increase; for TCP Reno we have k=0. The choice of k=0.8 is intended to make CTCP competitive with Highspeed TCP; we will return to the justification of this below. We will also choose α=1/8, which we will take as given. The value γ=30 here plays very roughly a similar role as Fast TCP's α, also 30, in that both represent a threshold for queue utilization.

When CTCP encounters a loss, we set

winsize = winsize$\times(1-\beta)$

While β is potentially configurable, typically we will have the usual β=1/2.

Finally we have the case where `diff` > γ; that is, the queue has grown "significantly". If `dwnd` is also positive, it is decremented. The variable `cwnd` continues to increase, but `cwnd` and `dwnd` will cancel each other out over the short term, leading to a roughly constant value for winsize. When `dwnd` drops to 0, however, this cancellation ends, and TCP Reno's `cwnd` += 1 per RTT takes over; `dwnd` has no more effect until after the next packet loss. Considering all these cases, a rough graph of the growth of CTCP's winsize is the following:



Compound TCP

We next derive k=0.8 as the value that leads to fair competition with Highspeed TCP. To do this we need a modest bit of calculus; the derivation can be skipped if preferred. We start with a hypothetical TCP

adjusting `cwnd` according to the rule `cwnd` += α × `cwnd`$^{0.8}$, per RTT, and show this TCP does indeed compete fairly with Highspeed TCP. If we measure time in RTTs, and denote `cwnd` by c = c(t), and extend c(t) to a continuous function of t, this increment rule becomes dc/dt = $\alpha \times c^{0.8}$. Taking reciprocals, we get dt/dc = $(1/\alpha) \times c^{-0.8}$. We can now integrate both sides, which yields $t = k_1 \times c^{0.2}$ (ignoring the constant of integration), or $c = k_2 \times t^5$. Integrating again, we get the number of packets in one tooth (the area) to be proportional to T^6, where T is the time at the right edge of the tooth. (We are inappropriately ignoring the left edge of the tooth, but by the argument of exercise 14.0 in *21.10 Exercises* this turns out not to matter.) This area is the reciprocal of the loss rate p. Solving for T, we get T proportional to $(1/p)^{1/6}$. As the average `cwnd` is proportional to T^5 (the area divided by T), by substitution we can conclude that `cwnd` is proportional to $p^{-5/6} = p^{-0.833}$ (versus the original exponent in *22.5 Highspeed TCP* of –0.835).

Calculating winsize$^{0.8}$ is hard to do rapidly, so in practice the exponent 0.75 is used. With that value the exponentiation can be done with two applications of a fast square-root algorithm.

CTCP turns out to compete reasonably fairly one-on-one with Highspeed TCP, by virtue of the choice of k=0.8. However, when competing with a set of TCP Reno connections, CTCP leaves the Reno connections with nearly the same bandwidth they would have had if they were competing with one more TCP Reno connection instead. That is, CTCP resists "stealing" bandwidth. CTCP does, however, make effective use of the bandwidth that TCP Reno leaves unclaimed due to the high-bandwidth TCP problem.

22.11 TCP Veno

TCP Veno ([FL03]) is a synthesis of TCP Vegas and TCP Reno, which attempts to use the RTT-monitoring ideas of TCP Vegas while at the same time remaining about as "aggressive" as TCP Reno in using queue capacity. TCP Veno has generally been presented as an option to address TCP's lossy-link problem, rather than the high-bandwidth problem *per se*.

A TCP Veno sender estimates the number N of packets likely in the bottleneck queue as N_{queue} = `cwnd` - BWE×RTT_{noLoad}, like TCP Vegas. TCP Veno then modifies the TCP Reno congestion-avoidance rule as follows, where the parameter β, representing the queue-utilization value at which TCP Veno slows down, might be around 5.

> if $N_{queue} < \beta$, `cwnd` = `cwnd` + 1 each RTT
> if $N_{queue} \geqslant \beta$, `cwnd` = `cwnd` + 0.5 each RTT

The above strategy makes `cwnd` growth less aggressive once link saturation is reached, but does continue to increase `cwnd` (half as fast as TCP Reno) until the queue is full and congestive losses occur.

When a packet loss does occur, TCP Veno uses its current value of N_{queue} to attempt to distinguish between non-congestive and congestive loss, as follows:

> if $N_{queue} < \beta$, the loss is probably *not* due to congestion; set `cwnd` = (4/5)×`cwnd`
> if $N_{queue} \geqslant \beta$, the loss probably *is* due to congestion; set `cwnd` = (1/2)×`cwnd` as usual

The idea here is that most router queues will have a total capacity much larger than β, so a loss with fewer than β likely does not represent a queue overflow. Note that, by comparison, TCP Westwood leaves `cwnd` unchanged if it thinks the loss is not due to congestion, and its threshold for making that determination is N_{queue}=0.

If TCP Veno encounters a series of non-congestive losses, the above rules make it behave like AIMD(1,0.8).

Per the analysis in *21.4 AIMD Revisited*, this is equivalent to AIMD(2,0.5); this means TCP Veno will be about twice as aggressive as TCP Reno in recovering from non-congestive losses. This would provide a definite improvement in lossy-link situations with modest bandwidth×delay products, but may not be enough to make a major dent in the high-bandwidth problem.

22.12 TCP Hybla

TCP Hybla ([CF04]) has one very specific focus: to address the TCP satellite problem (*4.4.2 Satellite Internet*) of very long RTTs. TCP Hybla selects a more-or-less arbitrary "reference" RTT, called RTT_0, and attempts to scale TCP Reno so as to behave like a TCP Reno connection with an RTT of RTT_0. In the paper [CF04] the authors suggest RTT_0 = 25ms.

Suppose a TCP Reno connection has, at a loss event at time t_0, reduced `cwnd` to $\texttt{cwnd}_{min}$. TCP Reno will then increment `cwnd` by 1 for each RTT, until the next loss event. This Reno behavior can be equivalently expressed in terms of the current time t as follows:

$$\texttt{cwnd} = (t - t_0)/RTT + \texttt{cwnd}_{min}$$

What TCP Hybla does is to use the above formula after replacing the actual RTT (or RTT_{noLoad}) with RTT_0. Equivalently, TCP Hybla defines the ratio of the two RTTs as $\varrho = RTT/RTT_0$, and then after each windowful (each time interval of length RTT) increments `cwnd` by ϱ^2 instead of by 1. In the event that $RTT < RTT_0$, ϱ is set to 1, so that short-RTT connections are not penalized.

Because `cwnd` now increases each RTT by ϱ^2, which can be relatively large, there is a good chance that when the network ceiling is reached there will be a burst of losses of size $\sim\varrho^2$. Therefore, TCP Hybla strongly recommends that the receiving end support SACK TCP, so as to allow faster recovery from multiple packet losses. Another recommended feature is the use of TCP Timestamps; this is a standard TCP option that allows the sender to include its own timestamp in each data packet. The receiver is to echo back the timestamp in the corresponding ACK, thus allowing more accurate measurement by the receiver of the actual RTT.

Finally, to further avoid having these relatively large increments to `cwnd` result in multiple packet losses, TCP Hybla recommends some form of **pacing** to smooth out the actual transmission times. Rather than sending out four packets upon receipt of an ACK, for example, we might estimate the time T to the next transmission batch (*eg* when the next ACK arrives) and send the packets at intervals of T/4. At the time TCP Hybla was developed, pacing was poorly supported, but see *22.16 TCP BBR* below, where pacing is essential.

TCP Hybla applies a similar ϱ-fold scaling mechanism to threshold slow start, when a value for `ssthresh` is known. But the initial unbounded slow start is much more difficult to scale. Scaling at that point would mean repeatedly doubling `cwnd` and sending out flights of packets, *before* any ACKs have been received; this would likely soon lead to congestive collapse.

22.13 DCTCP

Unlike the other TCP flavors in this chapter, **Data Center TCP** (DCTCP) is intended for use only by connections starting and ending within the same datacenter. DCTCP is *not* meant to be used on the Internet at large,

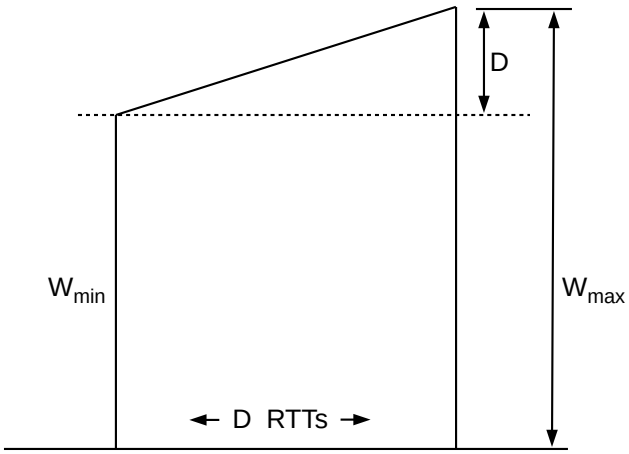
as it makes no pretense of competing fairly with TCP Reno. DCTCP was first described in [AGMPS10], and is now also specified in **RFC 8257**.

A datacenter is a highly specialized networking environment. Round-trip times on the Internet at large might be 50-100 ms, but round-trip times in a datacenter are usually well under 1 ms. Communicating nodes in a datacenter are under common management, and so there is no "chicken and egg" problem regarding software installation: if a new TCP feature is desired, it can be made available everywhere. Finally, cost-saving is an issue: datacenters have lots of switches and routers, and cheaper models generally have smaller queue capacities. Even with a 1-ms RTT, though, a 10 Gbps connection can have a bandwidth×delay product of 1.25 MB (800 packets); we would like to have queues be much smaller than this.

Recall that TCP Reno can be categorized as AIMD(1,0.5) (*21.4 AIMD Revisited*). The basic idea of DCTCP is to use AIMD(1,β) for values of β much smaller than 0.5. As the window-size reduction on packet loss is 1–β, this means that `cwnd` is relatively constant. If the transit capacity of a path is M, then the queue space needed to keep the minimum `cwnd` at M (and thus to keep the bottleneck link 100% utilized) is M×β/(1-β) $\simeq$ M×β if $\beta \simeq 0$.

This small β comes at the price of out-competing TCP Reno by a large margin. By Exercise 3.0 of *21.10 Exercises*, AIMD(1,β) is equivalent in terms of fairness to AIMD(α,0.5) for $\alpha = (2-\beta)/3\beta$, and by the argument in *20.3.1 Example 2: Faster additive increase* an AIMD(α,0.5) connection out-competes TCP Reno by a factor of α. For $\beta = 1/8$ we have $\alpha = 5$. For connections within a datacenter we can achieve fairness by implementing DCTCP everywhere, but introduction of DCTCP in the outside world DCTCP would be highly uncooperative.

The next step is to specify β. For the moment, we will make a simplifying assumption that there is only one connection, and no other traffic; in this case, the queue utilization increases by 1 for each RTT (once the queue becomes nonempty). We now determine β dynamically: we simply count the number D of RTTs before the queue is sufficiently full, and let $\beta = 1/2D$. (In [AGMPS10] and **RFC 8257**, 1/D is denoted by α, making $\beta = \alpha/2$, but to avoid confusion with the α in AIMD(α,β) we will write out the DCTCP α as `alpha` when we return to it below.)



DCTCP tooth

We can now relate D to `cwnd` and to the amplitude of `cwnd` variation. Let W_{max} denote the maximum `cwnd`, and W_{min} the minimum. Making the usual large-window simplifying assumptions, we have

$W_{min} + D = W_{max}$, because `cwnd` increases by 1 each RTT
$W_{min} = W_{max} \times (1-1/2D)$

Eliminating W_{min} and solving, we get $W_{max} = 2D^2$, or $D = \sqrt{}(W_{max}/2)$. Note that D is also the amplitude of the queue variation, assuming we keep the bottleneck link saturated, and so is the absolute minimum queue capacity needed. If the goal is keeping the queue small, this compares quite favorably to TCP Reno, in which $D = W_{min} = W_{max}/2$.

Now let K represent the maximum queue capacity; the next step is to relate K and D. We need to ensure that we can avoid having K be much larger than D. We have $W_{max} = TC + K$, where TC is the transit capacity of the link, that is, bandwidth×delay. We can express the minimum queue utilization as $Q_{min} = K - D = K - \sqrt{}((TC+K)/2$. If we choose K = TC, which is necessary with TCP Reno to avoid underutilized bandwidth, we certainly will have K much larger than D. However, to ensure $Q_{min} \geqslant 0$ we need $K = \sqrt{}((TC+K)/2$, or $K^2 = TC/2 + K/2$, which, because TC is relatively large (perhaps 800 packets), simply requires K just a bit larger than $\sqrt{}(TC/2)$. That is, K need not be much larger than D. At this point, we can afford to be more concerned with K's being too small, and thereby allowing intervals where the bottleneck link is idle.

Empirically, a workable value for the queue capacity K is around 65 for 10 Gbps Ethernet, which is moderately above $\sqrt{}(TC/2)$ but still very affordable. It is large enough that link utilization remains near 100%.

We now need to address the simplifying assumption that there was only one connection. First, there might be N connections, quite possibly synchronized. This means that the queue variation is N×D. It also means D will be somewhat smaller, though, as the total `cwnd` will be increasing N times faster.

A more serious issue is that there is also a *lot* of other traffic in a datacenter, so much so that queue utilization is dominated by a more-or-less random component. Instead of measuring when the queue utilization reaches a set level, we must measure when the *average* utilization reaches that level. DCTCP achieves this with a clever application of ECN (*21.5.3 Explicit Congestion Notification (ECN)*). The use of ECN to detect queue fullness, rather than packet drops, has the added advantage of avoiding packet losses and timeouts. Within a datacenter, DCTCP may very well rely on switch-based ECN rather than router-based.

In normal ECN, once the receiver has seen a packet with the CE bit, it is supposed to mark ECE (CE echo) in all returning ACKs until the sender acknowledges having responded to the congestion through the use of the CWR bit. DCTCP modifies this by having the receiver mark *only* ACKs to packets that arrive with CE set. This allows the sender to gauge the severity of congestion: if every other data packet has its CE bit set, then half the returning ACKs will be marked. (Delayed ACKs may complicate this, as the two data packets being acknowledged may have different CE marks, but mostly this is both infrequent and not serious, and in any event DCTCP recommends sending two separate ACKs with different ECE marks in such a case.)

Classically, having every other data packet marked should never happen: all data packets arriving at the router before the queue capacity K is reached should be unmarked, and all packets arriving after K is reached should be marked. But due to the random queue fluctuations described in the previous paragraph, this all-unmarked-then-all-marked pattern may be riddled with exceptions. What the DCTCP sender does is to measure the *average* marking rate, using an averaging interval at least as long as one "tooth". If there are D–1 unmarked RTTs and 1 marked RTT, then the average marking rate should be 1/D.

This is exactly what DCTCP looks for: once a significant number of marked ACKs arrives, indicating that congestion is experienced, the DCTCP sender looks at its running average of the marked fraction, and takes that to be 1/D. (More precisely, DCTCP denotes by `alpha` the marked fraction, and sets D = 1/`alpha`, and then $\beta = 1/2D =$ `alpha`/2.) DCTCP then reduces its `cwnd` by 1–β as above. The actual algorithm does not involve the queue capacity K, as a TCP sender is unlikely to know K.

While it is not part of DCTCP proper, another common configuration choice for intra-data-center connections is to reduce the minimum TCP retransmission timeout (RTO). The RTO value is computed adaptively,

as in *18.12 TCP Timeout and Retransmission*, but is subject to a minimum. As late as 2011, **RFC 6298** recommended (but did not require) a minimum RTO of 1.0 seconds, which is three orders of magnitude too large for a datacenter.

There is no global Linux minimum-RTO configuration setting, but this can be altered on a per-destination basis using the `ip route` command:

```
ip route change to 10.1.2.0/24 via 10.0.2.1 dev eth0 rto_min 20ms
```

The actual RTO values of current TCP connections can be viewed using the Linux command `ss --info`. On recent versions of Windows, a global minimum RTO can be set, for the `custom` template, using `netsh interface tcp set supplemental template=custom minRto=20`

22.13.1 TCP Incast

There is one particular congestion issue, mostly but not entirely exclusive to datacenters, that DCTCP does not handle directly: the **TCP incast** problem. Imagine one node sending out multiple simultaneous queries to "helper" nodes, and expecting more-or-less-simultaneous responses. One example might be a request for a large data block that has been distributed over multiple file-server systems; another might be a MapReduce request for calculation results. Either way, all the respondents may reply at about the same time, and all the responses may arrive together at the router and lead to queue overflow and packet loss. DCTCP (and any other TCP) cannot be of much help if each individual connection may be sending only one packet. This is one reason for having a slightly larger queue capacity than the DCTCP analysis alone might suggest.

The TCP incast problem is made much worse when (as is often the case) the helper-node requests must be executed serially; we saw this issue before with RPC in *16.5.3 Serial Execution*. Sometimes serialization requirements can be eliminated through careful design; sometimes they cannot.

22.14 H-TCP

H-TCP, or TCP-Hamilton, is described in [LSL05]. Like Highspeed-TCP it primarily allows for faster growth of `cwnd`; unlike Highspeed-TCP, the `cwnd` increment is determined not by the size of `cwnd` but by the elapsed time since the previous loss event. The threshold for accelerated `cwnd` growth is generally set to be 1.0 seconds after the most recent loss event. Using an RTT of 50 ms, that amounts to 20 RTTs, suggesting that when $\texttt{cwnd}_{min}$ is less than 20 then H-TCP behaves very much like TCP Reno.

The specific H-TCP acceleration rule first defines a time threshold t_L. If t is the elapsed time in seconds since the previous loss event, then for $t \leqslant t_L$ the per-RTT window-increment α is 1. However, for $t > t_L$ we define

$$\alpha(t) = 1 + 10(t - t_L) + (t - t_L)^2/4$$

We then increment `cwnd` by $\alpha(t)$ after each RTT, or, equivalently, by $\alpha(t)/\texttt{cwnd}$ after each received ACK.

At $t = t_L + 1$ seconds (nominally 2 seconds), α is 12. The quadratic term dominates the linear term when $t - t_L > 40$. If RTT = 50 ms, that is 800 RTTs.

Even if `cwnd` is very large, growth is at the same rate as for TCP Reno until $t > t_L$; one consequence of this is that, at least in the first second after a loss event, H-TCP competes fairly with TCP Reno, in the sense that

both increase `cwnd` at the same absolute rate. H-TCP starts "from scratch" after each packet loss, and does not re-enter its "high-speed" mode, even if `cwnd` is large, until after time t_L.

A full H-TCP implementation also adjusts the multiplicative factor β as follows (the paper [LSL05] uses β to represent what we denote by $1-\beta$). The RTT is monitored, as with TCP Vegas. However, the RTT increase is not used for per-packet or per-RTT adjustments; instead, these measurements are used after each loss event to update β so as to have

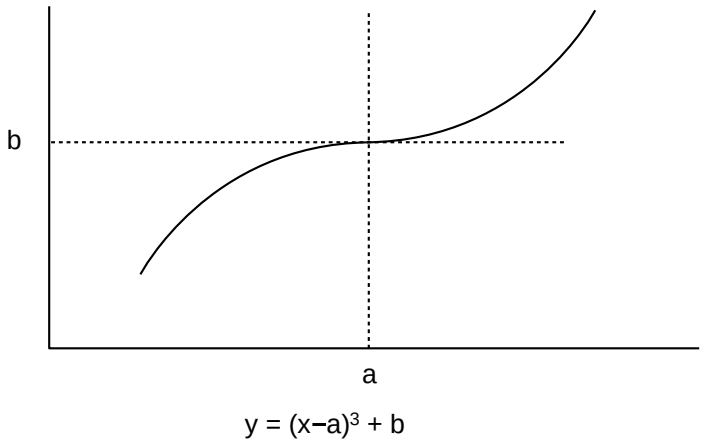
$$1-\beta = RTT_{min}/RTT_{max}$$

The value $1-\beta$ is capped at a maximum of 0.8, and at a minimum of 0.5. To see where the ratio above comes from, first note that RTT_{min} is the usual stand-in for RTT_{noLoad}, and RTT_{max} is, of course, the RTT when the bottleneck queue is full. Therefore, by the reasoning in *8.3.2 RTT Calculations*, equation 5, $1-\beta$ is the ratio transit_capacity / (transit_capacity + queue_capacity). At a congestion event involving a single uncontested flow we have `cwnd` = transit_capacity + queue_capacity, and so after reducing `cwnd` to $(1-\beta)\times$`cwnd`, we have $\texttt{cwnd}_{new}$ = transit_capacity, and hence (as in *19.7 TCP and Bottleneck Link Utilization*) the bottleneck link will remain 100% utilized after a loss. The cap on $1-\beta$ of 0.8 means that if the queue capacity is smaller than a quarter of the transit capacity then the bottleneck link *will* experience some idle moments.

When β is changed, H-TCP also adjusts α to $\alpha' = 2\beta\alpha(t)$ so as to improve fairness with other H-TCP connections with different current values of β.

22.15 TCP CUBIC

TCP Cubic attempts, like Highspeed TCP, to solve the problem of efficient TCP transport when bandwidth×delay is large. TCP Cubic allows very fast window expansion; however, it also makes attempts to slow the growth of `cwnd` sharply as `cwnd` approaches the current network ceiling, and to treat other TCP connections fairly. Part of TCP Cubic's strategy to achieve this is for the window-growth function to slow down (become concave) as the previous network ceiling is approached, and then to increase rapidly again (become convex) if this ceiling is surpassed without losses. This concave-then-convex behavior mimics the graph of the cubic polynomial `cwnd` = t^3, hence the name (TCP Cubic also improves an earlier TCP version known as TCP BIC).



As mentioned above, TCP Cubic is currently (2013) the default Linux congestion-control implementation. TCP Cubic was originally documented in [HRX08]; it is now specified in **RFC 8312**.

TCP Cubic has a number of interrelated features, in an attempt to address several TCP issues:

- Reduction in RTT bias
- TCP Friendliness when most appropriate
- Rapid recovery of `cwnd` following its decrease due to a loss event, maximizing throughput
- Optimization for an unchanged network ceiling (corresponding to $\texttt{cwnd}_{max}$)
- Rapid expansion of `cwnd` when a raised network ceiling is detected

The eponymous cubic polynomial $y=x^3$, appropriately shifted and scaled, is used to determine changes in `cwnd`. No special algebraic properties of this polynomial are used; the point is that the curve, while steadily increasing, is first concave and then convex; the authors of [HRX08] write "[t]he choice for a cubic function is incidental and out of convenience". This $y=x^3$ polynomial has an *inflection point* at x=0 where the tangent line is horizontal; this is the point where the graph changes from concave to convex.

We start with the basic outline of TCP Cubic and then consider some of the bells and whistles. We assume a loss has just occurred, and let W_{max} denote the value of `cwnd` at the point when the loss was discovered. TCP Cubic then sets `cwnd` to $0.7 \times W_{max}$; that is, TCP Cubic uses $\beta = 0.3$ (originally β was 0.2, but it has been adjusted to improve convergence to equilibrium between two flows). The corresponding α for TCP-Friendly AIMD(α,β) would be α=0.529, but TCP Cubic uses this α only in its TCP-Friendly adjustment, below.

We now define a cubic polynomial W(t), a shifted and scaled version of $w=t^3$. The parameter t represents the elapsed time since the most recent loss, in seconds. At time t>0 we set `cwnd` = W(t). The polynomial W(t), and thus the `cwnd` rate of increase, as in TCP Hybla, *is no longer tied to the connection's RTT*; this is done to reduce the RTT bias that is so deeply ingrained in TCP Reno.

We want the function W(t) to pass through the point representing the `cwnd` just after the loss, that is, $\langle t,W \rangle = \langle 0, 0.7 \times W_{max} \rangle$. We also want the inflection point to lie on the horizontal line $y=W_{max}$. To fully determine the curve, it is at this point sufficient to specify the value of t at this inflection point; that is, how far horizontally W(t) must be stretched. This horizontal distance from t=0 to the inflection point is represented by the constant K in the following equation; W(t) returns to its pre-loss value W_{max} at t=K. C is a second constant.

$$W(t) = C \times (t-K)^3 + W_{max}$$

It suffices algebraically to specify either C or K; the two constants are related by the equation obtained by plugging in t=0. K changes with each loss event, but it turns out that the value of C can be constant, not only for any one connection but for all TCP Cubic connections. TCP Cubic specifies for C the *ad hoc* value 0.4; we can then set t=0 and, with a bit of algebra, solve to obtain

$$K = (W_{max}/2)^{1/3} \text{ seconds}$$

If W_{max} = 250, for example, K=5; if RTT = 100 ms, this is 50 RTTs.

When each ACK arrives, TCP Cubic records the arrival time t, calculates W(t), and sets `cwnd` = W(t). At the next packet loss the parameters of W(t) are updated.

If the network ceiling does not change, the next packet loss will occur when `cwnd` again reaches the same W_{max}; that is, at time t=K after the previous loss. As t approaches K and the value of `cwnd` approaches W_{max}, the curve W(t) flattens out, so `cwnd` increases slowly.

This concavity of the cubic curve, increasing rapidly but flattening near W_{max}, achieves two things. First, throughput is boosted by keeping `cwnd` close to the available path transit capacity. In *19.7 TCP and Bottleneck Link Utilization* we argued that if the path transit capacity is large compared to the bottleneck queue capacity (and this is the case for which TCP Cubic was designed), then TCP Reno averages 75% utilization of the available bandwidth. The bandwidth utilization increases linearly from 50% just after a loss event to 100% just before the next loss. In TCP Cubic, the initial rapid rise in `cwnd` following a loss means that the average will be much closer to 100%. Another important advantage of the flattening is that when `cwnd` is finally incremented to the point of loss, it likely is just over the network ceiling; the connection has an excellent chance that only one or two packets are lost rather than a large burst. This facilitates the NewReno Fast Recovery algorithm, which TCP Cubic still uses if the receiver does not support SACK TCP.

Once t>K, W(t) becomes convex, and in fact begins to increase rapidly. In this region, `cwnd` > W_{max}, and so the sender knows that the network ceiling has increased since the previous loss. The TCP Cubic strategy here is to probe aggressively for additional capacity, increasing `cwnd` very rapidly until the new network ceiling is encountered. The cubic increase function is in fact quite aggressive when compared to any of the other TCP variants discussed here, and time will tell what strategy works best. As an example in which the TCP Cubic approach seems to pay off, let us suppose the current network ceiling is 2,000 packets, and then (because competing connections have ended) increases to 3,000. TCP Reno would take 1,000 RTTs for `cwnd` to reach the new ceiling, starting from 2,000; if one RTT is 50 ms that is 50 seconds. To find the time t-K that TCP Cubic will need to increase `cwnd` from 2,000 to 3,000, we solve $3000 = W(t) = C \times (t-K)^3 + 2000$, which works out to t-K $\simeq$ 13.57 seconds (recall 2000 = W(K) here).

The constant C=0.4 is determined empirically. The cubic inflection point occurs at $t = K = (W_{max} \times \beta/C)^{1/3}$. A larger C reduces the time K between the a loss event and the next inflection point, and thus the time between consecutive losses. If W_{max} = 2000, we get K=10 seconds when β=0.2 and C=0.4. If the RTT were 50 ms, 10 seconds would be 200 RTTs.

For TCP Reno, on the other hand, the interval between adjacent losses is $W_{max}/2$ RTTs. If we assume a specific value for the RTT, we can compare the Reno and Cubic time intervals between losses; for an RTT of 50 ms we get

W_{max}	Reno	Cubic
2000	50 sec	10 sec
250	6.2 sec	5 sec
54	1.35 sec	3 sec

For smaller RTTs, the basic TCP Cubic strategy above runs the risk of being at a competitive disadvantage compared to TCP Reno. For this reason, TCP Cubic makes a **TCP-Friendly adjustment** in the window-size calculation: on each arriving ACK, `cwnd` is set to the maximum of W(t) and the window size that TCP Reno would compute. The TCP Reno calculation can be based on an actual count of incoming ACKs, or be based on the formula $(1-\beta) \times W_{max} + \alpha \times t/RTT$.

Note that this adjustment is only "half-friendly": it guarantees that TCP Cubic will not choose a window size smaller than TCP Reno's, but places no restraints on the choice of a larger window size. Broadly speaking, however, in the range of smaller bandwidth×delay products where TCP Reno performs well, TCP Cubic relies on its TCP-Friendly adjustment to keep up; that is, its "native" window increase is less aggressive than

TCP Reno's. TCP Cubic is only supposed to be more aggressive than TCP Reno in settings where the latter is not aggressive enough.

A consequence of the TCP-Friendly adjustment is that, on networks with modest bandwidth×delay products, TCP Cubic behaves exactly like TCP Reno.

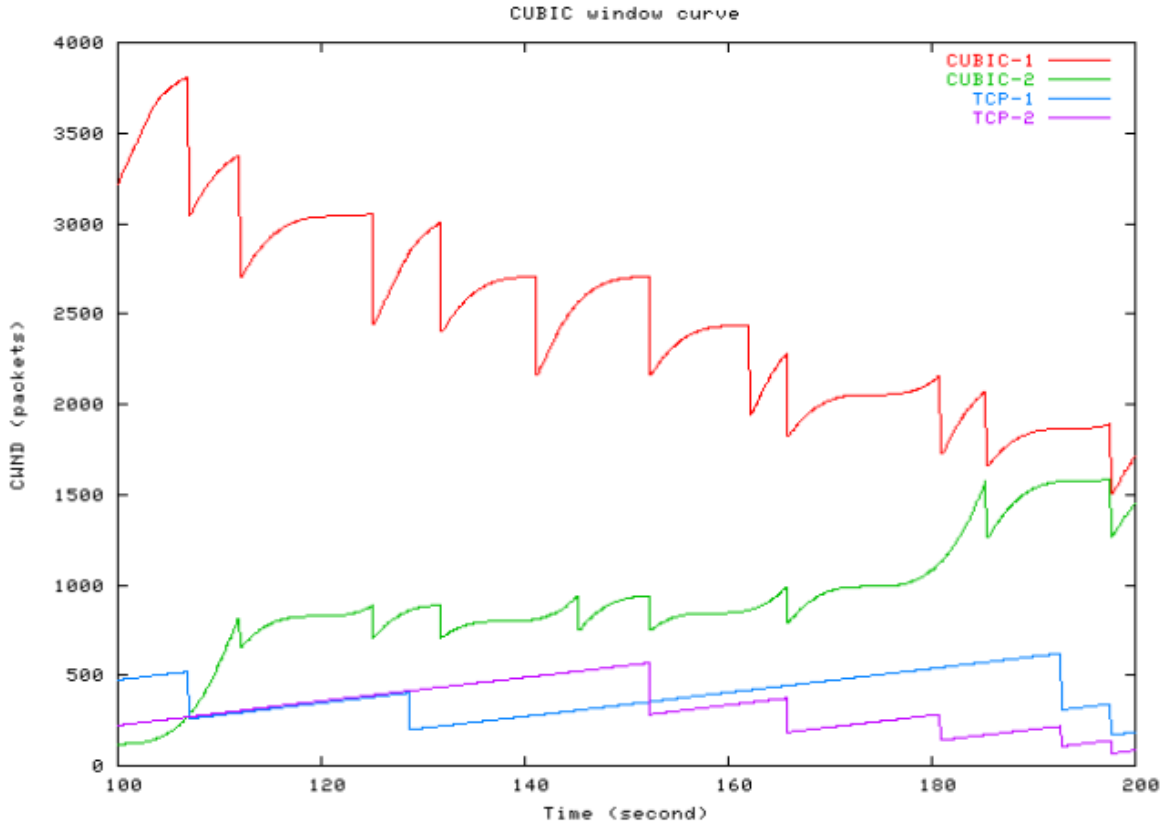
TCP Cubic also has a provision to detect if a given W_{max} is *lower* than the previous value, suggesting increasing congestion; in this situation, `cwnd` is lowered by an additional factor of $1-\beta/2$. This is known as **fast convergence**, and helps TCP Cubic adapt more quickly to reductions in available bandwidth.

Another way to gain a sense of how TCP Cubic compares with TCP Reno is to look at the rate of `cwnd` increase, per unit time. If W is the current window size, we can express this rate as $\Delta W/\Delta t$. For TCP Reno, if we take Δt to be the time between successive ACKs (as determined by the bottleneck bandwidth), ΔW is 1/W. For TCP Cubic, we will estimate $\Delta W/\Delta t$ by the derivative (from calculus) dW/dt of the formula $W(t) = C\times(t-K)^3 + W_{max}$, as above. This means $dW/dt = 3C(t-K)^2$. In order to get numbers we can actually compare, we need to look at specific scenarios in which we can evaluate K. At the left edge of the Cubic tooth, t=0.

For the first scenario, suppose the bandwidth is 1 packet/ms, and the delay is 50 ms, making the bandwidth×delay product 50 packets. We will also assume a queue capacity of 50 packets, so W_{max} = 100. This means that for TCP Reno, W increases by 1/50 per ms at the left edge of the tooth, where W = 50; the window growth rate $\Delta W/\Delta t$ is then (1/50)/(1/1000) = 20 packets/sec. At the right edge of the tooth, where W = 100, $\Delta W/\Delta t$ = (1/100)/(1/1000) = 10 packets/sec. For TCP Cubic, we use the formula above, and recall C = 0.4, so $\Delta W/\Delta t \simeq 1.2(t-K)^2$. At the left edge of the tooth, where t=0, this evaluates to $1.2K^2$. Recalling that $K = (W_{max}/2)^{1/3}$, we get K=3.68, and so $\Delta W/\Delta t \simeq 16$ packets/sec. This is comparable to TCP Reno. (At the inflection point of the cubic curve, though, where t=K, we always get $\Delta W/\Delta t = 0$.)

Now let's switch to a second, higher-bandwidth scenario, where the bottleneck bandwidth is increased to 100 packets/ms, the delay is again 50 ms, and the queue capacity is 3000 packets. The bandwidth×delay product is now 5000 packets, and so W_{max} = 5000 + 3000 = 8,000. As before, for TCP Reno $\Delta W/\Delta t$ ranges from (1/4,000)/(1/100,000) = 25 at the left edge of the tooth to 12.5 at the right edge; that is, the rate of `cwnd` growth is not much different from what it was in the first scenario. For TCP Cubic, on the other hand, we now have $K = 4000^{1/3} \simeq 16$, and so, at the left edge of the tooth where t=0, we have $\Delta W/\Delta t \simeq 1.2(K)^2 \simeq 300$. That is, `cwnd` is increasing at a rate of 300 packets/sec, which is twelve times more aggressive than TCP Reno.

The following graph is taken from [RX05], and shows TCP Cubic connections competing with each other and with TCP Reno.



The diagram shows four connections, all with the same RTT. Two are TCP Cubic and two are TCP Reno. The red connection, cubic-1, was established and with a maximum `cwnd` of about 4000 packets when the other three connections started. Over the course of 200 seconds the two TCP Cubic connections reach a fair equilibrium; the two TCP Reno connections reach a reasonably fair equilibrium with one another, but it is much lower than that of the TCP Cubic connections.

On the other hand, here is a graph from [LSM07], showing the result of competition between two flows using an earlier version of TCP Cubic over a low-speed connection. One connection has an RTT of 160ms and the other has an RTT a tenth that. The bottleneck bandwidth is 1 Mbit/sec, meaning that the bandwidth×delay product for the 160ms connection is 13-20 packets (depending on the packet size used).

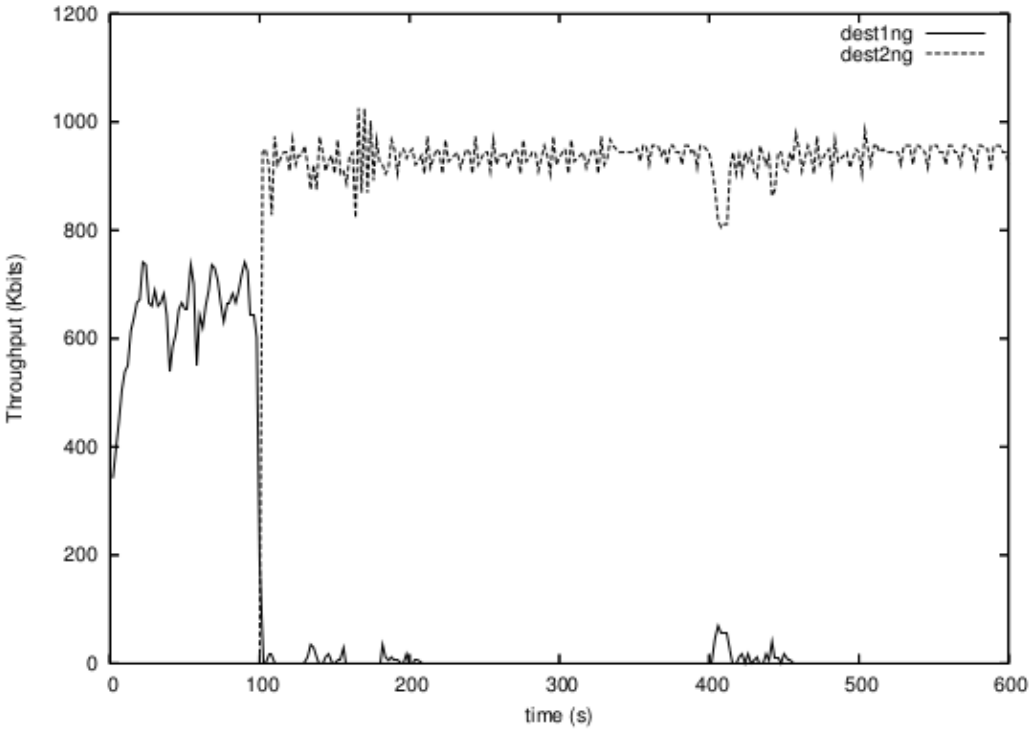


Fig. 14. Two Cubic TCP flows. 1Mbps link, flow 1 RTT 160ms, flow 2 RTT 16ms.

Note that the longer-RTT connection (the solid line) is almost completely starved, once the shorter-RTT connection starts up at T=100. This is admittedly an extreme case, and there have been more recent fixes to TCP Cubic, but it does serve as an example of the need for testing a wide variety of competition scenarios.

22.16 TCP BBR

TCP BBR returns to the central idea of TCP Vegas: to measure the available bandwidth and RTT_{min}, and to base the number of in-flight packets on the measured bandwidth×delay product. "BBR" here stands for **B**ottleneck **B**andwidth and **R**TT; it is described in [CGYJ16] and in an Internet Draft. There are some large differences from TCP Vegas, however; ultimately, these differences enable TCP BBR to compete reasonably fairly with TCP Reno. One important difference is that TCP BBR does not engage in the high-precision monitoring of RTT for increases above RTT_{noLoad}. As a result, TCP BBR does not fit the TCP Vegas delay-based congestion-control model; it is for that reason sometimes referred to as **congestion-based** congestion control.

TCP BBR is, in practice, **rate-based** rather than window-based; that is, at any one time, TCP BBR sends at a given calculated **rate**, instead of sending new data in direct response to each received ACK. Each arriving ACK does potentially update the current rate, much as each arriving ACK in TCP Reno slides the sender's window forwards; however, the connection between arriving ACKs and new data transmissions is decidedly indirect.

Rate-based sending requires some form of **pacing support** by the underlying LAN layer, so that packets

can be sent at equal time intervals. On a 10 Gbps link, this time interval can be as small as a microsecond; conventional timers don't work well at these time scales. Linux TCP BBR implementations generally use the pacing support built into the so-called Fair Queuing (FQ) queuing discipline (which is not actually a true Fair Queuing implementation in the sense of *23.5 Fair Queuing*).

Throughout the lifetime of a connection, TCP BBR maintains an estimate for RTT_{min}, which is nominally the stand-in for RTT_{noLoad} except that it may go up in the presence of competition; see below. TCP BBR also maintains a current bandwidth estimate, which we denote BWE. As with TCP Vegas, BWE is much more volatile than RTT_{min} as it better reflects the current degree of bandwidth competition. After each RTT, TCP BBR records the throughput during that RTT; BWE is then the maximum of the last ten per-RTT throughput measurements. That BWE is the *maximum* rate recorded over the past ten RTTs, rather than the *average*, will be important below.

The fundamental congestion indicators for TCP BBR are changes to its BWE and RTT_{min} estimates; packet losses are not used directly as evidence of congestion. As we shall see below, TCP BBR reduces its sending rate in response to decreases in BWE; this is TCP BBR's primary congestion response. When losses do occur, TCP BBR does enter a recovery mode, but it is much less conservative than TCP Reno's halving of `cwnd`. TCP BBR's initial response to a loss is to limit the number of packets in flight (FlightSize) to the number currently in flight, which allows it to continue to send new data at the rate of arriving ACKs. This is not necessarily a reduction in FlightSize, and, if it is, FlightSize may be allowed to grow, even if additional losses are discovered. Overall, this strategy is quite effective at handling non-congestive losses without losing throughput.

In its core state, known as **PROBE_BW**, TCP BBR continually updates BWE as above and then sets its base sending rate to BWE. It then sets its `cwnd` target (or, more properly, its FlightSize target, as losses may have occurred) to $2 \times BWE \times RTT_{min}$. This results in a bottleneck queue utilization equal to the transit capacity. If the actual available bandwidth does not change, then sending at rate BWE will send new packets at exactly the rate of returning ACKs and so FlightSize will not change. TCP BBR does allow for faster initial growth (see STARTUP mode, below) to reach the FlightSize target.

If the actual available bandwidth falls, BWE will not reflect that for ten RTTs. As a result, TCP BBR may for a while send faster than the rate of returning ACKs. If this happens, the bottleneck queue utilization will rise. Eventually, BWE will fall to match the rate of returning ACKs. Similarly, if the actual available bandwidth rises, queue utilization will fall. However, it will not fall to zero – and so cause sending to starve – in a single RTT unless the bandwidth doubles, and after that the increased bandwidth will be reflected in the updated BWE.

TCP BBR must, like every TCP flavor, regularly probe to see if additional bandwidth is available. TCP BBR does this by periodically (currently every eight RTTs, where RTT is measured as RTT_{min}) increasing its sending rate by an additional factor of 1.25; that is, it sets a variable `pacing_gain` to 1.25 and sends at the new rate `pacing_gain` $\times$ BWE. The increase lasts one RTT interval. *If* there was no competition, and if the bottleneck link was fully utilized, this `pacing_gain` increase results in no change to BWE. All that happens is that the queue builds up, and the 1.25-fold larger flight of packets results in an RTT that is also 1.25 times larger. In the next RTT interval, TCP BBR sets `pacing_gain` to 0.75, which causes the newly created additional queue to dissipate. After that it resumes its regular rate, that is, with `pacing_gain` = 1.0, for the next six RTT intervals.

Consider, however, what happens if TCP BBR is *competing*, perhaps with TCP Reno. Increasing the sending rate by a factor of 1.25 now results in greater queue (or bottleneck link) utilization, which results in an immediate increase in BWE for that RTT. At this point, recall that BWE is the maximum of the last ten per-RTT measurements; the end result is that BWE is set to this elevated value for the next ten RTTs. In

the following RTT, `pacing_gain` drops to 0.75 as before, but this time TCP BBR has measured a larger BWE, and this change to BWE persists.

Here is a concrete example of BWE increase. To simplify the analysis, we will assume TCP BBR's FlightSize is BWE $\times$ RTT_{min}, dropping the factor of 2. Suppose a TCP BBR connection and a TCP Reno connection share a bottleneck link with a bandwidth of 2 packets/ms. The RTT_{min} (= RTT_{noLoad}) of each connection is 80 ms, making the transit capacity 160 packets. Finally, suppose that each connection has 80 packets in flight, exactly filling the transit capacity but with no queue utilization (so RTT_{min} = RTT_{actual}). Over the course of the eight-RTT `pacing_gain` cycle, the Reno connection's `cwnd` rises by 8, to 88 packets. This means the total queue utilization is now 8 packets, divided on average between BBR and Reno in the proportion 80 to 88.

Now the BBR cycle with `pacing_gain`=1.25 arrives; for the next RTT, the BBR connection has 80×1.25 = 100 packets in flight. The total number of packets in flight is now 188. The RTT climbs to 188/2 = 94 ms, and the next BBR BWE measurement is 100 packets in 94 ms, or 1.064 packets/ms (the precise value may depend on exactly when the measurement is recorded). For the following RTT, `pacing_gain` drops to 0.75, but the higher BWE persists. For the rest of the `pacing_gain` cycle, TCP BBR calculates a base rate corresponding to 1.064 $\times$ 80 = 85 packets in flight per RTT, which is close to the TCP Reno `cwnd`. See also exercise 14.0.

TCP BBR also has another mechanism, arguably more important in the long run, for maintaining its fair share of the bandwidth. Periodically (every ~10 seconds), TCP BBR connections re-measure RTT_{min}, entering **PROBE_RTT** mode. In this state the number of packets in flight drops to four, and stays there for at least one RTT_{actual} as measured for these four packets (with a minimum of 200 ms). Afterwards the connection returns to PROBE_**BW** mode with a freshly estimated RTT_{min}. The value of BWE is picked up where it was left off, so that if RTT_{min} increases, then so does the sending rate BWE $\times$ RTT_{min}. A certain amount of potential throughput is "wasted" during these PROBE_RTT intervals, but as they amount to ~200 ms out of every 10 sec, or 2%, the impact is negligible.

If, during the PROBE_RTT mode, competing connections keep some packets in the bottleneck queue, then the queuing delay corresponding to those packets will be incorporated into the new RTT_{min} measurement; because of this, RTT_{min} may significantly exceed RTT_{noLoad} and thus cause TCP BBR to send at a more competitive rate. Suppose, for example, that in the BBR-vs-Reno scenario above, Reno has gobbled up a total of 240 spots in the bottleneck queue, thus increasing the RTT for both connections to (240+80)/2 = 160. During a PROBE_RTT cycle, TCP BBR will drop its link utilization essentially to zero, but TCP Reno will still have 240 packets in transit, so TCP BBR will measure RTT_{min} as 240/2 = 120 ms. After the PROBE_RTT phase is over, TCP BBR will increase its sending rate by 50% over what it had been when RTT_{min} was 80.

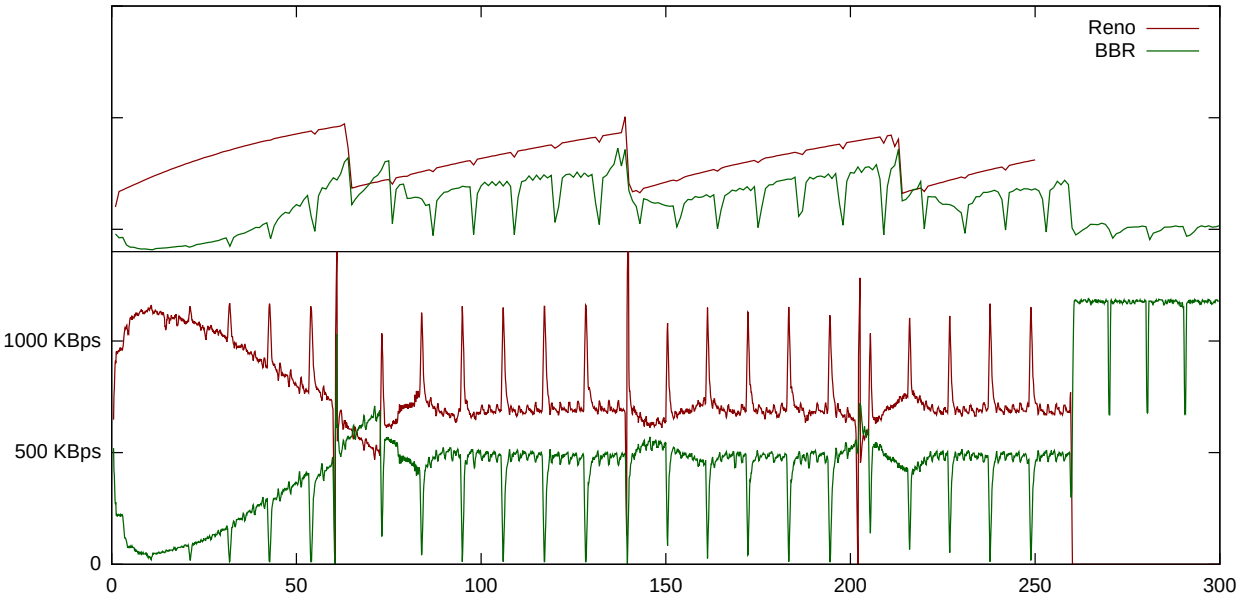
Note that, in any one RTT, we can either measure bottleneck bandwidth *or* RTT, but not both. If the number of packets in flight is larger than the transit capacity then the packet return rate reflects the bottleneck bandwidth. Conversely, we can measure RTT_{min} only if the number of packets in flight is smaller than the transit capacity.

When a connection is first opened, a TCP BBR connection is in **STARTUP** mode, which is similar to TCP Reno's slow start. In this mode, `pacing_gain` is 2.89 (2/log(2)) consistently, which leads to exponential growth of the number of packets in flight. STARTUP mode ends when an additional RTT yields no improvement in BWE. At this point TCP BBR has overfilled the queue substantially (just as a TCP Reno connection does in slow start), and so the connection enters **DRAIN** mode to reduce the queue. This is accomplished by setting `pacing_gain` = 1/2.89. The connection transitions from DRAIN to PROBE_RTT when the number of packets in flight drops to 2 $\times$ BWE $\times$ RTT_{min}.

Below is a diagram of TCP BBR competing with TCP Reno in a setting where the bottleneck queue capacity is eight times the bandwidth×delay product, which is 40 ms × 10 Mbps = 50 KB. It was produced using the Mininet network emulator, *30.7.6 TCP Competition: Reno vs BBR*. The large queue capacity was contrived specifically to be beneficial to TCP Reno, in that in a similar setting with a queue capacity approximately equal to the bandwidth×delay product TCP BBR often ends up quite a bit ahead of TCP Reno. Such large queues are, however, a not-uncommon real-world situation on high-capacity backbone links (*21.5.1 Bufferbloat*). Acting alone, Reno's `cwnd` would range between 4.5 and 9 times the bandwidth×delay product, which works out to keeping the queue over 70% full on average.

The lower part of the diagram shows each connection's share of the 10 Mbps (1.25 MBps) bottleneck bandwidth. The upper part shows the number of packets "in flight" (for TCP Reno, outside of Fast Recovery, that is of course `cwnd`). The Reno sawtooth pattern is clearly visible.

A dominant feature of the graph is the spikes every 10 seconds (down for BBR, correspondingly up for Reno) caused by TCP BBR's periodic PROBE_RTT mode.



For the first ten seconds, TCP Reno does indeed run away with all the bandwidth. But after the first PROBE_RTT event TCP BBR begins to catch up, and the two tie at around T=60 seconds. After that Reno mostly stays a little ahead of TCP BBR, typically with about 58% of the bandwidth versus BBR's 42%, but the point here is that, even in circumstances favorable to Reno, BBR does not collapse.

It is evident from the graph, particularly during the first 60 seconds, that the PROBE_RTT intervals do not lead to sudden jumps in throughput. Almost all of the change in throughput occurs during the PROBE_BW intervals. That said, it is the PROBE_RTT interval at T=10 that triggers the ensuing turnaround in throughput.

In addition to the sharp PROBE_RTT spikes every 10 seconds, we also see smaller spikes at a rate of about 6 every 10 seconds. These represent the pacing-gain cycling within BBR's PROBE_BW phase. If eight RTT_{min} times amount to 10/6 seconds, then RTT_{min} must be about 200 ms. When the queue is completely full, RTT_{actual} is 9 × 40 ms = 360 ms, but during TCP BBR's PROBE_RTT cycles RTT_{actual} does indeed drop considerably, which accounts for the 200 ms value. This value is then used as RTT_{min} for the next ten seconds.

Experimental results in [CGYJ16] indicate that TCP BBR has been much more successful than TCP Cubic in addressing the high-bandwidth TCP problem on parts of Google's network. This is presumably because TCP BBR does not necessarily reduce throughput at all when faced with occasional non-congestive losses.

22.17 Epilog

TCP Reno's core congestion algorithm is based on algorithms in Jacobson and Karel's 1988 paper [JK88], now (2017) approaching thirty years old. There are concerns both that TCP Reno uses too much bandwidth (the greediness issue) and that it does not use enough (the high-bandwidth-TCP problem).

There are also broad changes in TCP usage patterns. Twenty years ago, the vast majority of all TCP traffic represented downloads from "major" servers. Today, over half of all Internet TCP traffic is peer-to-peer rather than server-to-client. The rise in online video streaming creates new demands for excellent TCP real-time performance.

So which TCP version to use? That depends on circumstances; some of the TCPs above are primarily intended for relatively specific environments; for example, TCP Hybla for satellite links and TCP Veno for mobile devices (including wireless laptops). If the sending and receiving hosts are under common management, and especially if intervening traffic patterns are relatively stable, one can run a few simple throughput-comparison experiments to find which TCP version works best.

But there are two problems with this experimental approach. First, intervening traffic patterns are often *not* stable; a TCP version that worked well in one traffic environment might perform poorly in another. TCP Vegas, after all, does well in a Vegas-only environment; problems arise only when there is competing TCP Reno traffic, or the equivalent. Second, and perhaps more seriously, the best-performing TCP version might achieve its throughput at the expense of other users' TCP traffic. As a simple example, consider the effect of simply increasing the TCP Reno additive-increase value, perhaps from AIMD(1,0.5) to AIMD(**10**,0.5). As we saw in *20.3.1 Example 2: Faster additive increase*, this gives the faster-incrementing TCP an unfair (in fact tenfold) advantage. If the goal is to find a TCP version that *all* users will be happy with, this will not be effective.

Then there is the question of what TCP to use on a server that is serving up large volumes of data, to a range of disparate hosts and with a wide variety of competing-traffic scenarios. Here, experimentation is even more difficult. Many trials will be needed to determine reliably which TCP version works best in the most cases, even ignoring the impact on competing traffic. These issues suggest a need for continued research into how to update and improve TCP, and Internet congestion-management generally.

Finally, while most new TCPs are designed to hold their own in a Reno world, there is some question that perhaps we would all be better off with a radical rather than incremental change. Might TCP Vegas be a better choice, if only the queue-grabbing greediness of TCP Reno could be restrained? Questions like these are today entirely hypothetical, but it is not impossible to envision an Internet backbone that implemented non-FIFO queuing mechanisms (*23 Queuing and Scheduling*) that fundamentally changed the rules of the game.

22.18 Exercises

1.0. How would TCP Vegas respond if it estimated RTT_{noLoad} = 100ms, with a bandwidth of 1 packet/ms, and then due to a routing change the RTT_{noLoad} increased to 200ms without changing the bandwidth? What `cwnd` would be chosen? Assume no competition from other senders.

2.0. Suppose a TCP Vegas connection from A to B passes through a bottleneck router R. The RTT_{noLoad} is 50 ms and the bottleneck bandwidth is 1 packet/ms.

(a). If the connection keeps 4 packets in the queue (*eg* α=3, β=5), what will RTT_{actual} be? What value of `cwnd` will the connection choose? What will be the value of BWE?

(b). Now suppose a competing (non-Vegas) connection keeps 6 packets in the queue to the Vegas connection's 4, eventually meaning that the other connection will have 60% of the bandwidth. What will be the Vegas connection's steady-state values for RTT_{actual}, `cwnd` and BWE?

3.0. Suppose a TCP Vegas connection has R as its bottleneck router. The transit capacity is M, and the queue utilization is currently Q>0 (meaning that the transit path is 100% utilized, although not necessarily by the TCP Vegas packets). The current TCP Vegas `cwnd` is $\texttt{cwnd}_V$. Using the formulas from *8.3.2 RTT Calculations*, show that the number of packets TCP Vegas calculates are in the queue, **queue_use**, is

queue_use = $\texttt{cwnd}_V \times Q/(Q+M)$

4.0. Suppose that at time T=0 a TCP Vegas connection and a TCP Reno connection share the same path, and each has 100 packets in the bottleneck queue, exactly filling the transit capacity of 200. TCP Vegas uses α=1, β=2. By the previous exercise, in any RTT with $\texttt{cwnd}_V$ TCP Vegas packets and $\texttt{cwnd}_R$ TCP Reno packets in flight and $\texttt{cwnd}_V+\texttt{cwnd}_R>200$, N_{queue} is $\texttt{cwnd}_V/(\texttt{cwnd}_V+\texttt{cwnd}_R)$ multiplied by the total queue utilization $\texttt{cwnd}_V+\texttt{cwnd}_R-200$.

Continue the following table, where T is measured in RTTs, up through the next two RTTs where $\texttt{cwnd}_V$ is *not* decremented; that is, find the next two rows where the TCP Vegas queue share is less than 2. (After each of these RTTs, $\texttt{cwnd}_V$ is not decremented.) This can be done either with a spreadsheet or by simple algebra. Note that the TCP Reno $\texttt{cwnd}_R$ will always increment.

T	$\texttt{cwnd}_V$	$\texttt{cwnd}_R$	TCP Vegas queue share
0	100	100	0
1	101	101	1
2	102	102	2
3	101	103	(101/204)x4 = 1.980 < β
4	101	104	Vegas has (101/205)×5 = 2.463 packets in queue
5	100	105	Vegas has (100/205)×5 = 2.435 packets in queue
6	99	106	(99/205)×5 = 2.439

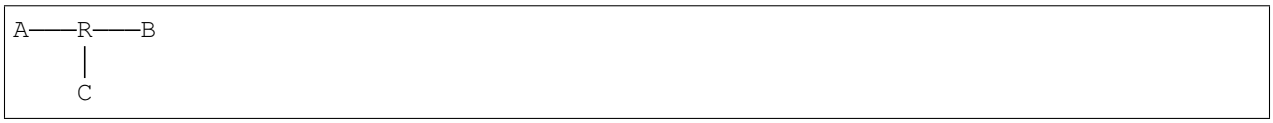
This exercise attempts to explain the *linear decrease* in the TCP Vegas graph in the diagram in *31.5 TCP Reno versus TCP Vegas*. Competition with TCP Reno means not only that $\texttt{cwnd}_V$ stops increasing, but in fact it decreases by 1 most RTTs.

5.0. Suppose that, as in the previous exercise, a FAST TCP connection and a TCP Reno connection share the same path, and at T=0 each has 100 packets in the bottleneck queue, exactly filling the transit capacity of 200. The FAST TCP parameter γ is 0.5. The FAST TCP and TCP Reno connections have respective cwnds of $\texttt{cwnd}_F$ and $\texttt{cwnd}_R$. You may use the fact that, as long as the queue is nonempty, $\text{RTT}/\text{RTT}_{\text{noLoad}} = (\texttt{cwnd}_F+\texttt{cwnd}_R)/200$.

Find the value of $\texttt{cwnd}_F$ at T=40, where T is counted in units of 20 ms until T = 40, using α=4, α=10 and α=30. Assume RTT $\simeq$ 20 ms as well. Use of a spreadsheet is recommended. The table here uses α=10.

T	$\texttt{cwnd}_F$	$\texttt{cwnd}_R$
0	100	100
1	105	101
2	108.47	102
3	110.77	103
4	112.20	104

6.0. Suppose A sends to B as in the layout below. The packet size is 1 kB and the bandwidth of the bottleneck R–B link is 1 packet / 10 ms; returning ACKs are thus normally spaced 10 ms apart. The $\text{RTT}_{\text{noLoad}}$ for the A–B path is 200 ms.



However, large amounts of traffic are also being sent from C to A; the bottleneck link for that path is R–A with bandwidth 1 kB / 5 ms. The queue at R for the R–A link has a capacity of 40 kB. ACKs are 50 bytes.

(a). What is the maximum possible arrival time difference on the A–B path for ACK[0] and ACK[20], if there are no queuing delays at R in the A→B direction? ACK[0] should be forwarded immediately by R; ACK[20] should have to wait for 40 kB at R

(b). What is the minimum possible arrival time difference for the same ACK[0] and ACK[20]?

7.0. Suppose a TCP Veno and a TCP Reno connection compete along the same path; there is no other traffic. Both start at the same time with cwnds of 50; the total transit capacity is 160. Both share the next loss event. The bottleneck router's queue capacity is 60 packets; sometimes the queue fills and at other times it is empty. TCP Veno's parameter β is zero, meaning that it shifts to a slower cwnd increment as soon as the queue just begins filling.

(a). In how many RTTs will the queue begin filling?

(b). At the point the queue is completely filled, how much larger will the Reno cwnd be than the Veno cwnd?

8.0. Suppose two connections use TCP Hybla. They do not compete. The first connection has an RTT of 100 ms, and the second has an RTT of 1000 ms. Both start with $\texttt{cwnd}_{\text{min}}$ = 0 (literally meaning that nothing

is sent the first RTT).

(a). How many packets are sent by each connection in four RTTs (involving three `cwnd` increases)?
(b). How many packets are sent by each connection in four seconds? Recall 1+2+3+...+N = N(N+1)/2.

9.0. Suppose that at time T=0 a TCP Illinois connection and a TCP Reno connection share the same path, and each has 100 packets in the bottleneck queue, exactly filling the transit capacity of 200. The respective `cwnd`s are $\texttt{cwnd}_\text{I}$ and $\texttt{cwnd}_\text{R}$. The bottleneck queue capacity is 100.

Find the value of $\texttt{cwnd}_\text{I}$ at T=50, where T is the number of elapsed RTTs. At this point $\texttt{cwnd}_\text{R}$ is, of course, 150.

T	$\texttt{cwnd}_\text{I}$	$\texttt{cwnd}_\text{R}$
0	100	100
1	101	101
2	?	102

You may assume that the delay, RTT – RTT_noLoad, is proportional to queue_utilization = $\texttt{cwnd}_\text{I}$+$\texttt{cwnd}_\text{R}$−200α. Using this expression to represent delay, delay_max = 100 and so $\text{delay}_\text{thresh}$ = 1. When calculating α(delay), assume α_max = 10 and α_min = 0.1.

10.0. Assume that a TCP connection has an RTT of 50 ms, and the time between loss events is 10 seconds.

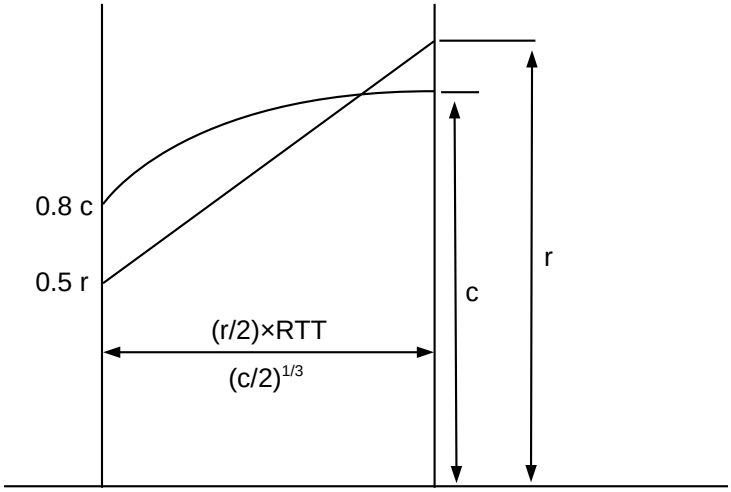
(a). For a TCP Reno connection, what is the bandwidth×delay product?
(b). For an H-TCP connection, what is the bandwidth×delay product?

11.0. For each of the values of W_max below, find the *change* in TCP Cubic's `cwnd` over one 100 ms RTT at each of the following points:

i. Immediately after the previous loss event, when t = 0.
ii. At the midpoint of the tooth, when t=K/2
iii. At the point when `cwnd` has returned to W_max, at t=K

(a). W_max = 250 (making K=5)
(b). W_max = 2000 (making K=10)

12.0. Suppose a TCP Reno connection is competing with a TCP Cubic connection. There is no other traffic. All losses are synchronized. In this setting, once the steady state is reached, the `cwnd` graphs for one tooth will look like this:



One tooth, TCP Cubic v TCP Reno

Let c be the maximum `cwnd` of the TCP Cubic connection ($c=W_{max}$) and let r be the maximum of the TCP Reno connection. Let M be the network ceiling, so a loss occurs when c+r reaches M. The width of the tooth for TCP Reno is $(r/2)\times$RTT, where RTT is measured in seconds; the width of the TCP Cubic tooth is $(c/2)^{1/3}$. For the examples here, ignore the TCP-Friendly feature of TCP Cubic.

(a). If M = 200 and RTT = 50 ms = 0.05 sec, show that at the steady state $r \simeq 130.4$ and $c = M-r \simeq 69.6$.
(b). Find equilibrium r and c (to the nearest integer) for M=1000 and RTT = 50 ms. Hint: use of a spreadsheet or scripting language makes trial-and-error quite practical.
(c). Find equilibrium r and c for M = 1000 and RTT = 100 ms.

13.0. Suppose a TCP Westwood connection has the path A——R1——R2——B. The R1–R2 link is the bottleneck, with bandwidth 1 packet/ms, and RTT_{noLoad} is 200 ms. At T=0, with `cwnd` = 300 so the queue at R1 has 100 A–B packets, the R1–R2 throughput for A's packets falls to 1 packet / 2 ms, perhaps due to competition. At that same time, and perhaps also due to competition, a single A–B packet is lost at R1.

(a). Suppose A responds to the loss using the original BWE of 1 packet/ms. What transit capacity will A calculate, and how will A update its `cwnd`?

(b). Now suppose A uses the new throughput of 1 packet / 2 ms as its BWE. What transit capacity will A calculate, and how will A update its `cwnd`?

(c). Suppose A calculates BWE as `cwnd`/RTT. What value of BWE does A obtain by measuring the RTT of the packet just before the one that was lost?

14.0. In *22.16 TCP BBR* we estimated the impact on TCP BBR's BWE value during the interval when `pacing_gain`=1.25. Suppose now that the BBR and Reno connections each have 800 packets in transit, instead of 80. Assume the bottleneck bandwidth rises tenfold to 20 packets/ms, so RTT_{noLoad} is still 80 ms. During the 8-RTT pacing-gain cycle, Reno increases its `cwnd` to 808. *If* BWE is measured at the optimum point after BBR's `pacing_gain`=1.25 rate increase, what is the new value of BWE?